Conceptual Modeling of a Quantum Key Distribution Simulation Framework Using the Discrete Event System Specification

DISSERTATION

Jeffrey D. Morris, Master Sergeant, USA

AFIT-ENV-DS-14-S-25

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States

AFIT-ENV-DS-14-S-25

CONCEPTUAL MODELING OF A QUANTUM KEY DISTRIBUTION SIMULATION FRAMEWORK USING THE DISCRETE EVENT SYSTEM SPECIFICATION

DISSERTATION

Presented to the Faculty

Department of Systems Engineering and Management

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

Jeffrey D. Morris, BS, MMIS, MSSI

Master Sergeant, USA

September 2014

DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENV-DS-14-S-25

CONCEPTUAL MODELING OF A QUANTUM KEY DISTRIBUTION SIMULATION FRAMEWORK USING THE DISCRETE EVENT SYSTEM SPECIFICATION

Jeffrey D. Morris, BS, MMIS, MSSI
Master Sergeant, USA

Approved:

__________//signed//__________________________ 25 Aug 2014
Michael R. Grimaila, PhD, CISM, CISSP (Chairman) Date

_________//signed//___________________________ 19 Aug 2014
Douglas D. Hodson, PhD (Member) Date

________//signed//____________________________ 25 Aug 2014
David R. Jacques, PhD (Member) Date

________//signed//____________________________ 15 Aug 2014
Gerry Baumgartner, PhD (Member) Date

Accepted:

____ ___________
ADEDEJI B. BADIRU Date
Dean, Graduate School of Engineering
and Management

AFIT/ENV/DS/14S-25

Abstract

Quantum Key Distribution (QKD) is a revolutionary security technology that exploits the laws of quantum mechanics to achieve information-theoretical secure key exchange. QKD is suitable for use in applications that require high security such as those found in certain commercial, governmental, and military domains. As QKD is a new technology, there is a need to develop a robust quantum communication modeling and simulation framework to support the analysis of QKD systems.

This dissertation presents conceptual modeling QKD system components using the Discrete Event System Specification (DEVS) formalism to assure the component models are provably composable and exhibit temporal behavior independent of the simulation environment. These attributes enable users to assemble and simulate any collection of compatible components to represent QKD system architectures. The developed models demonstrate *closure under coupling* and exhibit behavior suitable for the intended analytic purpose, thus improving the validity of the simulation.

This research contributes to the validity of the QKD simulation, increasing developer and user confidence in the correctness of the models and providing a composable, canonical basis for performance analysis efforts. The research supports the efficient modeling, simulation, and analysis of QKD systems when evaluating existing systems or developing next generation QKD cryptographic systems.

Acknowledgements

First and foremost, I would like to express my sincere thanks and gratitude to my advisor, Dr. Michael Grimaila, for his guidance and support through this effort. He offered me the opportunity to expand my horizons. Without his support, I would not have begun this journey and he was there to mentor me throughout, whether I needed an enlightening conversation or a push along the way. I would like to thank Dr. Douglas Hodson and the QKD project sponsors for their endless patience and gently correcting my misconceptions along the way. Finally, I need to thank Dr. Sarjoughian of ACIMS for his expertise and RTSync for use of their DEVS software.

I must thank my wonderful wife for her unflagging support during this experience. She was with me every step of this journey, handling everything life brought to us, so I could keep focus on my research. I could never have done this without her loving support.

Jeffrey D. Morris

Table of Contents

List of Figures

List of Tables

CONCEPTUAL MODELING OF A QUANTUM KEY DISTRIBUTION SIMULATION FRAMEWORK USING THE DISCRETE EVENT SYSTEM SPECIFICATION

1. Introduction

1.1 *The Need for Secure Communications*

CRYPTOGRAPHY, the practice and study of techniques for securing communications between two authorized parties in the presence of one or more unauthorized parties, is the centerpiece of a centuries old battle between code maker and code breaker (Singh, 1999). Historically, only financial, government, and military entities used cryptography; but today much of modern society depends on cryptography to provide security services including confidentiality, integrity, authentication, and non-repudiation (Barker, Barker, & Lee, 2005). While there are many types of cryptography, only the One-Time-Pad (OTP) symmetric key algorithm is "information-theoretically secure" (C. E. Shannon, 1948; C. E. Shannon, 1949). All other forms of cryptography are breakable if the adversary has enough cipher text, computational resources, and time (Schneier, 1995), either by finding flaws in the encryption algorithm or decoding the cipher text. Despite its strength, the OTP is not in common use because of the large amount of secret key material required for its proper use (i.e. random generation, length equal to the message, and single use)(Bellovin, 2011). These requirements impose significant limitations on use of the OTP in most applications due the costs involved with secure key generation and distribution.

Quantum Key Distribution (QKD) is a technology that offers the means for two geographically separated parties to generate a shared secret key (Grimaila, Morris, & Hodson, 2012). QKD is unique in its ability to detect any eavesdropping on the key exchange, assuring the secrecy of the key. This is possible due to the fundamental laws of quantum mechanics which ensures eavesdropping on the quantum channel introduces detectable errors. QKD enables an “unconditionally secure” cryptosystem when paired with the OTP.

1.2 *Quantum Key Distribution (QKD)*

In 1984, Charles Bennett and Gilles Brassard proposed the first QKD protocol, BB84, for secure communication (Bennett and Brassard, 1984). The goal of the system is to provide perfect secrecy during key distribution. Using a QKD protocol, a sender and receiver exchange an unconditionally secure secret key by leveraging properties of quantum mechanics. QKD enables two parties to “grow” a shared secret key without placing any limits on an adversary’s computational power and can detect the presence of any third-party eavesdropping on the key exchange. Because of the laws of quantum mechanics, any third-party eavesdropping on the key exchange will introduce detectable errors. If the errors are below a defined threshold, the two parties involved in the key exchange can distill an unconditionally secure key. QKD provides a potential solution to the key distribution problem by enabling two parties connected by a quantum channel to continuously produce an unconditionally secure shared secret key, or decide not to use a key if detecting an eavesdropper.

1.3 *The Need for QKD Simulation*

QKD is a developing technology and not thoroughly studied from a systems-level perspective. QKD systems contain non-ideal components that differ, sometimes significantly, from the ideal components specified during the original conceptual system design. Therefore, there is a need to develop an efficient integrated modeling and simulation capability to understand the impact non-ideal components have on the performance and security of different QKD system architectures.

Because of the limits of technology, it is impossible to build the ideal QKD system described in theory (Scarani et al., 2009). Therefore, each QKD implementation is only an approximation of the ideal apparatus described in theory. Therefore, our research effort focuses on the development of a QKD modeling and simulation framework that includes system implementation non-idealities in the system analysis to better understand their impact on overall system performance and security.

There exist few QKD simulations beyond those that model specific hardware or situations. An example is the Austrian Institute of Technology’s AIT QKD Software project (Austrian Institute of Technology, 2014) that attempts to model an entire QKD network but focuses on one QKD hardware type (entanglement). An extensive literature search over several years revealed no other multi-technology system-level QKD modeling & simulation (M&S) efforts.

To address this shortcoming, a research team at the Air Force Institute of Technology developed a modular simulation framework, named *qkdX*, which provides users the capability to model efficiently, simulate, and study QKD system architectures.

This simulation capability provides hybrid functionality as it abstracts continuous-time QKD system signals (e.g., electrical signals and optical pulses) into a representation suitable as events in a Discrete Event Simulation (DES) environment (Morris, Hodson, Grimaila, Jacques, & Baumgartner, 2014).

1.4 *Problem Description*

Just as in the early days of computing, each QKD system, whether commercial or research, is a unique implementation based on the theory and principles of QKD using currently available components, protocols, and technology. As there are no widely accepted security and performance standards for evaluating QKD systems, each system designer architects their system based on their own views and needs. The ability to model accurately and simulate QKD systems at an appropriate abstraction level is an essential capability necessary for analysis of current and next generation QKD cryptographic systems.

Currently, there exists no efficient means of modeling, simulating, and analyzing different QKD systems. There is a need to develop a flexible, extendable, quantum communication modeling and simulation analysis framework that take advantage of all the best practices in modeling, simulation, and analysis and model QKD systems at an appropriate detail level to estimate system-level attributes in areas such as security, performance, and cost.

Best practices in modeling and simulation provide the "simulation study" (Banks, Carson, Nelson, & Nicol, 2010) as an accepted approach in building models and simulations. Different versions of the simulation study exist, and can contain from 10-12 steps, but the focus of this research is on the first three steps (identifying the problem, setting the objectives, and conceptual modeling) using the version proposed by Banks (Banks & Gibson, 2001). Investigation into the first two steps of the simulation study and guidance from the QKD research sponsor lead to five research questions that provided insight necessary for the final two primary research questions.

Two issues in building any simulation are validation and verification. The first is a question of "did we model the right thing?" and the seconds is "did we build the right model?" The QKD simulation needs to answer these questions or risk non-acceptance from the end-users and project sponsors. To that end, this research proposes to use conceptual model validity theory and the DEVS modeling formalism to increase the validity of the simulation, and is the focus of the journal article in chapter 7.

1.4.1 *Research Focus & Methodology*

Creating a *conceptual model* is the third step of the simulation study and the primary focus of this research. This conceptual model is the bridge to link the mathematical models provided by the Subject Matter Experts (SMEs) to the computer code created by the project coders for the selected simulation framework. Conceptual modeling increased the validity of the simulation by using several well-accepted validation and verification (V&V) techniques and expressing the conceptual model using a well-documented, proven, modeling formalism.

The intent of this research is to explore these primary and secondary research objectives:

Primary objectives:

1. Develop conceptual models for each optical component identified for the prototypical QKD architecture
2. Use conceptual model validity theory to increase the validity of the simulation

Secondary research objectives:

1. Enumerate the elements contained in QKD systems.
2. Identify the requirements for a robust QKD modeling, simulation, and analysis capability.
3. Propose a simulation environment which meets the needs of the research.
4. Develop a prototypical QKD system architecture for analysis in the dissertation research.

This research focused on the proper modeling of the temporal behavior and internal “state” of optical components for creating conceptual models for use in the *qkdX* simulation. This researcher, in consultation with SMEs in the optical physics and electrical engineering domains, determined the necessary detail level for each model. These models enable a system-level simulation where signals propagate through the system as discrete events, but can be reconstructed into a continuous-time representation when requiring mathematical operations or transformations of the signals.

To capture the temporal behavior and the state of components, this researcher used DEVS. In the past, DEVS has been used to model high-level architectures, hybrid-systems, cell-spaces, distributed supply chains, test & evaluation, forest fires, environmental systems, building performance models, and other problem spaces (Gunay, O'Brien, Goldstein, Breslav, & Khan, 2013; Mittal, Risco, & Zeigler, 2007; Ntaimo, Zeigler, Vasconcelos, & Khargharia, 2004; G. A. Wainer & Giambiasi, 2001; G. Wainer, 2006; B. P. Zeigler, Kim, & Buckley, 1999; B. P. Zeigler, Song, Kim, & Praehofer, 1995; B. P. Zeigler, Ball, Cho, Lee, & Sarjoughian, 1999). This research represents, to the team’s knowledge, the first use of DEVS to model optical components.

1.5 *Document Organization*

This document uses a journal format to present work related to conceptual modeling of optical components using DEVS. Chapter 2 presents my research questions and discusses how the research questions link to four of the included articles. Additionally, it presents the methodology for testing the DEVS atomic and coupled models.

Chapter 3 is the article “Quantum Key Distribution: A Revolutionary Security Technology” published in the *ISSA Journal*, a trade publication of the Information Systems Security Association (Grimaila et al., 2012). This article presents a “layman’s” explanation of QKD technology and the BB84 protocol. It is included only as background information for better understanding QKD.

Chapter 4 presents the article “Towards Modeling and Simulation of Quantum Key Distribution Systems.” This article appeared in the International Journal of Emerging Technology and Advanced Engineering (Morris et al., 2014). It presents the research findings for several of the secondary research questions.

Chapter 5 is the article titled “A Survey of Quantum Key Distribution (QKD) Technologies.” The book *Emerging Trends in ICT Security* (Morris, Grimaila, Hodson, Jacques, & Baumgartner, 2013) contains this article as its chapter 9. This material highlights research into QKD technologies to identify optical components for the prototypical QKD architecture.

Chapter 6 contains the article titled “A Reference Architecture to Enable Security and Performance Analysis of Quantum Key Distribution." This article presents a discussion of the baseline reference QKD architecture used in the conceptual modeling research. It reviews some of the design decisions made for d the reference architecture and is under review at *IEEE Transactions on Emerging Topics in Computing*.

Chapter 7 is the article titled "Using the Discrete Event System Specification to Model Quantum Key Distribution System Components." This article is the capstone of the research using the DEVS framework. It includes a discussion on how the conceptual were built, the modeling process and how the research modeling efforts increased the validity of the *qkdX* simulation and is under review at the *Journal of Defence Modeling and Simulation*.

Chapter 8 documents the results and analysis of the conceptual modeling effort. It describes the modeling steps and lists the component and coupled submodules built during the research. It discusses testing the models and provides examples of model code and output data.

Chapter 9 presents the conclusions, significance of this work, and recommendations for further research. Chapter 10 contains the references for chapters 1-2, and 8-10. Each embedded article and the 27 appendices have its own reference list, focused on the material in that document.

Appendix A contain systems engineering material relating to the QKD prototypical architecture, a decomposition of the QKD system down to the individual optical components within the Alice quantum module and offers descriptions of the software tools used in this research. Appendix B contains information on the component creation and the testing methodology and has examples of testing output for both components and coupled submodules. Appendix C is a short primer on cryptography, the next seventeen contain the DEVS research documentation for each modeled optical component and the final seven appendices cover the modeled coupled submodules for the "Alice" portion of a QKD system. See Table 1 for a concise list.

Table 1. *List of Appendices.*

Appendix #	Title	Contents
A	QKD Prototypical Architecture	System Engineering & decomposition diagrams
B	Component Creation and Testing	Overview of creating & testing steps
C	Cryptography Overview	Primer on cryptography
D	Bandpass Filter	Component description, DEVS documentation & Use Cases
E	Beamsplitter	Component description, DEVS documentation & Use Cases
F	Circulator	Component description, DEVS documentation & Use Cases
G	Optical Photodiode (Classical Detector)	Component description, DEVS documentation & Use Cases
H	EVOA	Component description, DEVS documentation & Use Cases
I	Fixed Attenuator	Component description, DEVS documentation & Use Cases
J	Half-wave Plate	Component description, DEVS documentation & Use Cases
K	In-line Polarizer	Component description, DEVS documentation & Use Cases
L	Isolator	Component description, DEVS documentation & Use Cases
M	Laser	Component description, DEVS documentation & Use Cases
N	PM Fiber	Component description, DEVS documentation & Use Cases

O	Polarization Controller	Component description, DEVS documentation & Use Cases
P	Pulse Modulator	Component description, DEVS documentation & Use Cases
Q	Polarizing Beamsplitter	Component description, DEVS documentation & Use Cases
R	SM Fiber	Component description, DEVS documentation & Use Cases
S	Optical Switch	Component description, DEVS documentation & Use Cases
T	WDM	Component description, DEVS documentation & Use Cases
U	CPG Module	Submodule description, DEVS documentation & Use Cases
V	PM Module	Submodule description, DEVS documentation & Use Cases
W	DSG Module	Submodule description, DEVS documentation & Use Cases
X	CTQ Module	Submodule description, DEVS documentation & Use Cases
Y	OSL Module	Submodule description, DEVS documentation & Use Cases
Z	TPG Module	Submodule description, DEVS documentation & Use Cases
AA	OPM Module	Submodule description, DEVS documentation & Use Cases

2. Methodology

The purpose of this chapter is to present the research questions, their place within the overall research and the methodology used for the research. Each question has a short discussion and highlights the chapter containing the article focused on that research.

2.1 *Research Questions*

A search of simulation literature provided an accepted path for creating a simulation: the simulation study. In simulation literature, the accepted practice to approach problem solving is to use a methodical simulation study (Banks, 1998; Elliott, Edmondson, Scrudder, Igarza, & Smith, 2009; Geoffrion, 1989; Gogg & Mott, 1998; Jain, 1991; Naylor & Finger, 1967).

Simulation studies have many suggested steps, and Banks suggests a version of the simulation study supported in literature and drawn from research started in the 1960s (Banks & Gibson, 1997; Banks, 1998; Banks & Gibson, 2001; Banks & Chwif, 2010; Banks et al., 2010; Law, Kelton, & Kelton, 1991; R. E. Shannon, 1998). Banks writes extensively on this topic and the DOD MSCO references the process as a best practice (Morse et al., 2010). Banks' simulation study includes many steps but the scope of this research is his first three steps:

- Problem formulation
- Setting of objectives and overall project plan
- Model conceptualization

This research addressed these steps by posing multiple research questions. The questions explore the topics necessary to address the simulation study steps and meet the requests of the project sponsor. The intent was to conduct research to answer the questions, meet the needs of the project sponsor and support the research objectives of the AFIT QKD research team.

Five initial questions were posed after conducting a literature review and discussions with SMEs and project sponsor. Answering these questions laid the foundation for the research into the conceptual model for the QKD simulation. The five questions were:

2.1.1 *Q1: What Are the Basic Elements of a QKD System?*

This research focused on identifying the elements of QKD (components, architectures, phases and processes) and distilling a list of optical components selected for inclusion in the prototypical demonstration architecture. These components were the focus of the DEVS modeling research. The primer article in chapter 3 provides the basic understanding of QKD and the article in chapter 4 presents an overview of QKD technologies found during this research.

2.1.2 *Q2: What End User Capabilities are required in a QKD Modeling and Simulation Framework?*

Evaluating the needs of users served a dual purpose: 1) provided research into a topic requested by the project sponsor to ensure the QKD simulation provided capabilities relevant to the eventual end-users, 2) captured requirements for inclusion in the conceptual model and demonstration architecture. Chapter 5 contains the article discussing the research for this question.

2.1.3 *Q3: What Software Developer Capabilities are required in a QKD Modeling and Simulation Framework?*

The project sponsor also requested researching developer capabilities for the QKD simulation. These results lead to identifying additional requirements for inclusion in the conceptual models and the demonstration architecture. Chapter 5 contains the article discussing the research for this question.

2.1.4 *Q4: Which Simulation Environment Is Best for QKD Modeling and Simulation?*

This research was exploration and comparison of various simulation software packages to select the best simulation environment for building the QKD simulation. Both open-source and professional software packages were considered, leading to identifying the best solution, based on the capabilities identified in the earlier research questions and input from the project sponsor. Chapter 5 contains the article discussing the research for this question.

2.1.5 *Q5: What QKD System Architecture is Best Suited to Demonstrate the Analysis Capabilities of the Proposed Framework?*

This question addressed creating a QKD system architecture using the optical components identified by the first research question. A demonstration QKD simulation built by the AFIT research team modeled this architecture using the simulation environment identified in the previous question as a proof-of-concept project. Chapter 6 has the draft article discussing the *qkdX* simulator and the prototypical architecture. This article is still in the draft stage, with submission to the *IEEE Transactions on Emerging Topics in Computing* expected by mid-August.

2.1.6 *Q6: What is the DEVS formalism for each component within the prototypical QKD system?*

The purpose of this question is to use the DEVS formalism to describe the conceptual models for the selected prototypical QKD system per the 3rd step of the simulation study. The DEVS formalism provides a comprehensive description of the system in an accepted modeling and simulation formalism. The DEVS formalism aids the model developers, provides increased validation of the QKD simulation, and supports testing and evaluation. Chapter 7 is the article submitted to *The Journal of Defense Modeling and Simulation* for review and provides a discussion on the DEVS formalism for an example coupled submodule and an example optical component.

2.1.7 *Q7: How can we use the DEVS formalism and conceptual model validity theory to increase the validity of the QKD simulation?*

The purpose of this question is to mate the DEVS formalism to conceptual model theory to increase the perceived validity and confidence in the QKD simulation conceptual models by using theory, techniques, and principles of V&V. This was accomplished by applying several V&V techniques and providing the following list of products:

- English-language rules for each component and module
- Phase Transition Diagrams for each component and module
- Event-Trace Diagrams for each component and module
- SME math-based behavioral models for each component (SME input)
- DEVS pseudo code for each component and module

The research for each component starts with a description of the component derived from commercial data sheets and academic texts describing the components. This provided the necessary information to build a conceptual model for the component. Phase transition diagrams showed timing and behavior and event-trace diagrams, in the form of state list tables, described the transitions between states for several test cases. This information, with the SME-provided mathematical behavior models, became the basis for constructing the DEVS pseudocode.

DEVS provides a way to formalize the conceptual model of the QKD simulator, but in itself does not provide conceptual model validity. While verification and validation are normally used together, validation is the focus of this research. Model validity is a necessary condition for the credibility of simulation results (Balci, 1995). Model validation, according to Balci, is “substantiating that the simulation model, within its domain of applicability, behaves with satisfactory accuracy consistent with the study objectives” (Balci, 1997). Model validation is the comparison of model behavior to the behavior of the system under study when both are responding to identical input conditions (Sargent, 2005). The article in chapter 7 (pages 7-8) discusses conceptual modeling, DEVS, and how using DEVS increases the validity of a simulation.

2.2 *Methodology*

The methodology for creating the DEVS models for each component and coupled submodule is the same and the results for each are included in the appendices. Chapter 7 explains the methodology in detail and explains the research results. The basic procedure started with the optical physics SME creating a mathematical model for each of the optical components that captured parameters and behavior believed necessary for the model. Model creation used a combination of data measured during laboratory experiments and component data sheets and existing reference literature. Creating and verifying the correctness of the developed math models used Mathematica, a well-known math computation software package (Wolfram, 2014).

The next step was to transform the mathematical model into a DEVS pseudocode model. These mathematical models become the “source system” shown in Figure 1 and consequently, the basis for QKD simulator. The modeler reviewed the math models to understand the necessary transformation functions, reviewed quantum and optical physics literature and consulted with the SMEs to understand the required component behavior. Product literature for existing physical components provided additional information for acceptable component input and output ranges. The DEVS models captured this information using phases, states and transitions and submitted the component models back to the optical SME for review.

Once complete, the DEVS pseudocode became the basis for creating the model in a DEVS-compliant simulator, MS4ME (MS4 Systems, 2014). MS4ME is a product of RTSync (www.rtsync.com), a spin-off from the Arizona Center of Integrative Modeling and Simulation (ACIMS) (Arizona State University, 2014). MS4ME provides a structured user interface for modeling built on top of the DEVSJAVA simulator (B. P. Zeigler, Sarjoughian, & Au, 1997). For each component, the output from the MS4ME simulator was compared against the expected behavior of the DEVS model. This modeling was a check on the DEVS pseudocode and ensured the models met the requirements of the formalism and captured the appropriate behavior. Once checked, the DEVS pseudocode became the basis for the simulation modelers to create the *qkdX* framework.

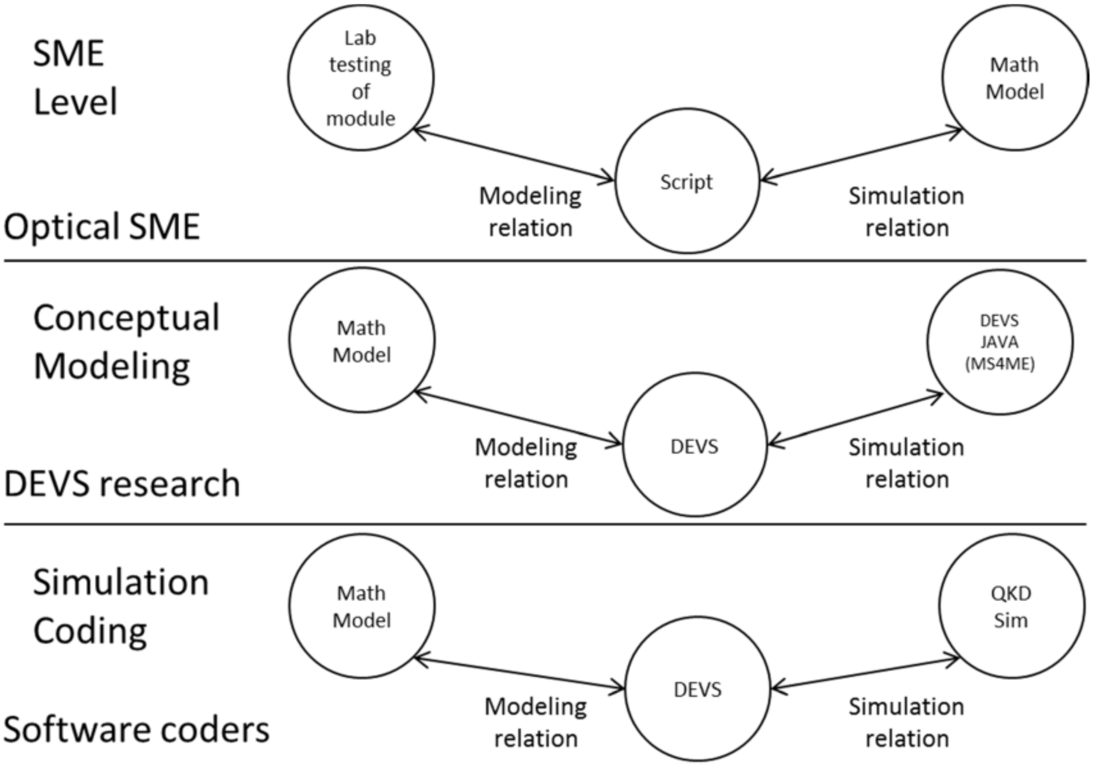


Figure 1. Levels of modeling and simulation.

2.2.1 *Component testing*

The components were tested within the MS4ME simulator, with the results shared with the SME for face validity and trace checking (see chapter 6 for explanation of these validation techniques), as well as using operational graphics (viewing the model as it moves through time) and fixed values (using constants for model input and internal variables to check against calculated values). These four techniques are a subset of the suggested validation techniques by Sargent for validation of simulation models (Sargent, 2005).

Appendix B describes the component creation and testing process, with a section on component behavior testing, a section with an example of the MS4ME output from a test case used for the EVOA, and section listing a “pseudocode” derived from code comments within the EVOA code.

The steps of the creating and testing process were:

- Component description – describing the component function and physical design using commercial and academic literature.
- Component conceptual model – text description of the properties and behaviors of interest in the component.
- English-language rules – list of rules that describe the behavior of the component.
- Phase transition diagram – diagram that shows how the component moves from phase to phase within each state. This diagram is described in detail in chapter 6.
- Event-trace diagram – diagrams and tables describing how the component moves from phase to phase for several test cases.
- Use case – list of use cases for the component.
- DEVS code – DEVS pseudocode for the component; used to create the MS4ME models.
- MS4ME code – programming the pseudocode into the MS4ME simulator as a check to ensure the pseudocode captured the behavior and timing properly.
- MS4ME output review – a line-by-line check of the MS4ME output to ensure the MS4ME model and the pseudocode matched and the MS4ME programming was correct.
- MS4ME model review – the optical SME reviewed the DEVS conceptual model along with the MS4ME models to ensure the modeled behavior and timing was correct in comparison to the starting mathematical model.
- Model refactor – after review, each model was corrected per feedback from the optical SME.

This is just a short overview of the process, see appendix B for a detailed description.

2.2.2 *Methodology Issues*

A major component of validating the DEVS pseudocode was using the MS4ME DEVS simulator, but several issues arose during its use. MS4ME is still in development and the researcher discovered issues that affected the research. The most important of these was the lack of advanced math functions in the DEVS-JAVA language that is the basis of MS4ME.

The design of optical packets uses integration functions to determine several values during simulation execution. This design required integrating a scientific library into the *qkdX* simulation framework, but the researcher found the MS4ME simulator failed when trying to install a comparative JAVA-language scientific library. This issue was reported to RTSync, but there was no fix to this issue during the research period.

This problem affected research during the coupled submodules building phase. Each time an optical packet reflects, there should be an optical power reduction calculated using the integration functions. The missing integration functions led to optical packets reflecting infinitely between components. Each component was tested individually for proper behavior by injecting low-power packets and ensuring the fiber components properly deleted these low-power packets per the simulation design choice to have the fiber modules delete packets below a specified power level.

The generally accepted method ensuring validity of a simulation is to compare the simulation output to a real-world system. The closer the outputs (or the harder it is to distinguish between the two), the stronger the validity. This method is not applicable when simulating future or notional systems, and in this research, not having access to real systems for comparison. Additionally, the prototypical architecture was intentionally created not to model any existing QKD system but rather to exercise the capabilities of *qkdX*.

To overcome these deficiencies, the researcher relied on guidance from an optical physics SME who provided the mathematical models for the optical components. The

SME reviewed each DEVS and MS4ME model for accuracy (as noted in the previous discussion on research methodology) to ensure they captured the proper behavior. Late into this research period, the AFIT QKD research team did receive a QKD system, but there was no time to capture data from this system.

2.2.3 *Included Articles*

The following five chapters contain articles focused on the various aspects of this research. Chapter 3 contains the first article which provides a layman's primer on QKD. Chapter 4 presents an overview of QKD technologies and networks, part of the research into identifying the components necessary to model QKD systems. Chapter 5 focuses on identifying requirements and capabilities for modeling QKD and selection of a simulation environment. Chapter 6 presents the *qkdX* simulation built by the AFIT research team and discusses the components and submodules necessary to model the demonstration QKD architecture. Finally, chapter 7 presents the research into DEVS, writing the DEVS pseudocode and how using DEVS can increase the validity of QKD simulation.

3. Quantum Key Distribution: A Revolutionary Security Technology

Grimaila, M. R., Morris, J. D., & Hodson, D. (2012). Quantum Key Distribution: A Revolutionary Security Technology. *The ISSA Journal, 27*, 20-27.

Published in: **ISSA Journal, Volume 27, June 2012**

12 Pages

Note: This article has been changed from its published format for inclusion in this document.

Quantum Key Distribution: A Revolutionary Security Technology

Michael R. Grimaila, Jeffrey Morris, and Douglas Hodson

Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765

Introduction

Quantum Key Distribution (QKD) is a revolutionary security technology that exploits the laws of quantum mechanics to achieve information-theoretic secure key exchange. QKD enables two parties to "grow" a shared secret key without placing any limits on an adversary's computational power. QKD is unique in its ability to detect the presence of any third-party eavesdropping on the key exchange. Due to the fundamental laws of quantum mechanics, any third-party eavesdropping on the key exchange will introduce detectable errors. If the errors are below a defined threshold, an unconditionally secure key can be distilled. When QKD is used in conjunction with the On-Time Pad symmetric cryptographic algorithm, the result is an unconditionally secure cryptographic system. In this article, we provide a brief background of cryptography related to QKD, present the basic principles the BB84 QKD protocol, and discuss vulnerabilities arising from the non-idealities present in real world QKD system implementations.

Secure Communications and Cryptography

The need for secure communications has existed since the dawn of humanity. Cryptography, the practice and study of techniques for securing communications between two authorized parties in the presence of one or more unauthorized third parties, is an essential tool used to assure information security (Rivest, 1990). Historically, government and military applications chiefly used cryptography, but today almost everyone is dependent on cryptography as it is used to provide security services including confidentiality, integrity, authentication, authorization, and non-repudiation (NIST 800-21, 1995).

A cryptosystem is composed of two basic components: an algorithm and one or more keys. The algorithm is the mathematical transformation used to encrypt and decrypt messages and the key(s) are parameters used in the encryption and decryption processes. Figure 1 shows a block diagram of a simple cryptosystem. The original message, *m*, called the "plaintext" transforms into the "ciphertext", $E_K(m)$, using the encryption algorithm, E, and the encryption key, K. The terms plaintext and ciphertext refer to binary data and can represent anything in digital form (e.g., text, audio, video, pictures, programs). Other parameters (e.g., initialization vectors, salt, etc.) may be used but are not shown for simplicity. Ideally, the ciphertext is not decipherable unless you possess the key required to decrypt it[1]. The transformation of plaintext into ciphertext "protects"

[1] This is not strictly true as the strength of most cryptographic algorithms is rooted in the effort required to solve difficult mathematical problems. The science of cryptanalysis is focused upon learning the secret key based upon analysis of the ciphertext, as well as other information related to the encryption and decryption

the confidentiality of messages transmitted over a public channel where an adversary can possibly intercept it. Upon receipt, the ciphertext message is transformed back into the plaintext, *m*, using the decryption algorithm, D, and the decryption key K′. The decryption algorithm, D, is the inverse transformation of the encryption algorithm, M, which means that $D_{K}′$ $(E_{K}(m))=m$. Note that in general K does not have to equal K′, although it does for symmetric algorithms.

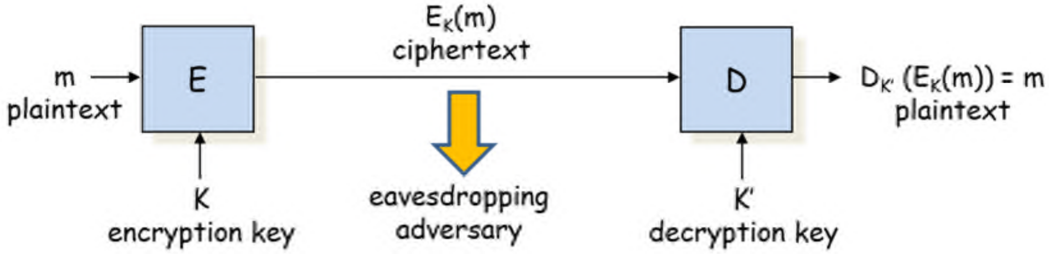


Figure 1 – A Simple Cryptosystem Block Diagram

There are three basics types of cryptographic algorithms: symmetric, asymmetric and hashing functions. Symmetric key algorithms use the same key (e.g., K = K′) for encryption and decryption. The benefits of a symmetric key algorithm are that it provides confidentiality, is fast, is easily implemented in hardware, and consumes little computational power when compared to asymmetric algorithms. However, symmetric key algorithms only provide confidentiality and require a separate key for each pair of entities who wish secure communications, which does not scale well when large numbers of entities must securely communicate. Examples of symmetric key algorithms include DES, 3-DES, AES, Blowfish, RC4, and RC5.

Asymmetric key algorithms used mathematically related, but different (e.g., K ≠ K′), key pairs for encryption and decryption (e.g., public and private keys) which reduces the key distribution burden. The benefits of an asymmetric key algorithm are that there no need for “out of band” key distribution as public keys are freely shared, it scales better since only a single key pair needed for each individual, and additionally provides authentication and non-repudiation. However, asymmetric key algorithms are slow because they require complex mathematical operations and typically consume more computational power than symmetric algorithms. Examples of asymmetric algorithms include RSA, PGP, El Gamal, ECC, and Diffie-Hellman.

Hash algorithms are one-way functions that take an arbitrary length message and create a fixed length message digest. Hash algorithms may or may not require a key depending on their mode of operation. Hashing provides an efficient way to check the integrity of stored or transmitted data without having to compare the data bit by bit. However, since hash algorithms map all possible inputs to a fixed length message digest, more than one

process. As technology progresses, an adversary with unlimited computational resources may be able to decode the ciphertext in a “reasonable” amount of time. There are concerns that certain cryptographic algorithms will become useless when quantum computers with a larger number of qbits become viable.

input can map to the same digest creating a “collision” which may provide an advantage to an eavesdropper. Examples of hash algorithms include MD-4, MD-5, and SHA-1.

Different cryptographic algorithms can be used independently or can be combined in a hybrid fashion to provide robust security services. For example, a web browser that interacts with a secure web server using SSL/TLS uses all three of the primary cryptographic algorithms.

The One-Time Pad (OTP)

The only cryptographic algorithm mathematically proven to be unconditionally secure is the One-Time Pad (OTP). The OTP is a symmetric cryptographic algorithm and is relatively easy to understand. The first known description of the OTP was in 1882 when Frank Miller described “superencipherment” as a means to insure the privacy and secrecy of telegraphic communications (Bellovin, 2011). Miller’s method required the use of a randomly generated key that was never reused. In 1917, Gilbert Vernam invented and later patented a cipher based on teleprinter technology, but it was vulnerable because it reused key material (Vernam, 1919; Vernam, 1926). Despite this weakness, a National Security Agency report identified Vernam’s patent as “perhaps one of the most important in the history of cryptography” (Kahn, 1996). Subsequently, Joseph Mauborgne recognized that if the key used in the Vernam cipher was fully random, then cryptanalysis would be impossible. In the 1940s, Claude Shannon proved the theoretical significance of the security of the OTP (Shannon, 1949).

The strength of all other modern cryptographic algorithms is based upon “computational security,” which means that it is considered secure if there is a negligible probability of determining the key in a “reasonable” amount of time using current technology. In theory, every cryptographic algorithm, except the OTP, is insecure given enough ciphertext, computational resources, and time[2].

To use the OTP, each of the parties (common called Alice and Bob) who wish secure communications must first share a key "pad" which is a randomly generated key, equal in length to the message to be sent[3]. Alice transforms the plaintext into the ciphertext by bitwise Exclusive-ORing (XORing) the plaintext with the key pad. The XOR operation of two bits is calculated as follows: if the bits are the same you will obtain a `0`, and if the bits are different you will obtain a `1`. Once Alice has encrypted the plaintext into the ciphertext, she destroys the key pad and transmits the ciphertext message. Upon receiving the message, Bob XORs the ciphertext message with the same key pad used by Alice to recover the plaintext message and then destroys his key pad.

[2] Recent developments in quantum computing have placed the security of certain cryptographic algorithms (e.g., RSA) which are based upon the difficulty of factoring large numbers into their constituent primes at risk.

[3] This simple requirement is the most limiting factor when using the OTP as you must have a virtually infinite supply of a shared secret key available to Alice and Bob.

For example, suppose Alice wants to securely send a message (1001) to Bob using the OTP as shown in Figure 2. First, Alice and Bob must preshare a secret key pad (0101) that is the same length as the message. Next, Alice XORs the plaintext message (1001) with the key (0101) to obtain the ciphertext (1100). Alice now sends the ciphertext through the public insecure network. When Bob receives the ciphertext (1100), he XORs it with the key pad (0101) to recover the plaintext (1001). One way to think about the OTP is that the key pad is "noise" that mixes with the plaintext to create the ciphertext. Since only Alice and Bob have the same noise source (key pad), they are the only ones who can filter it out.

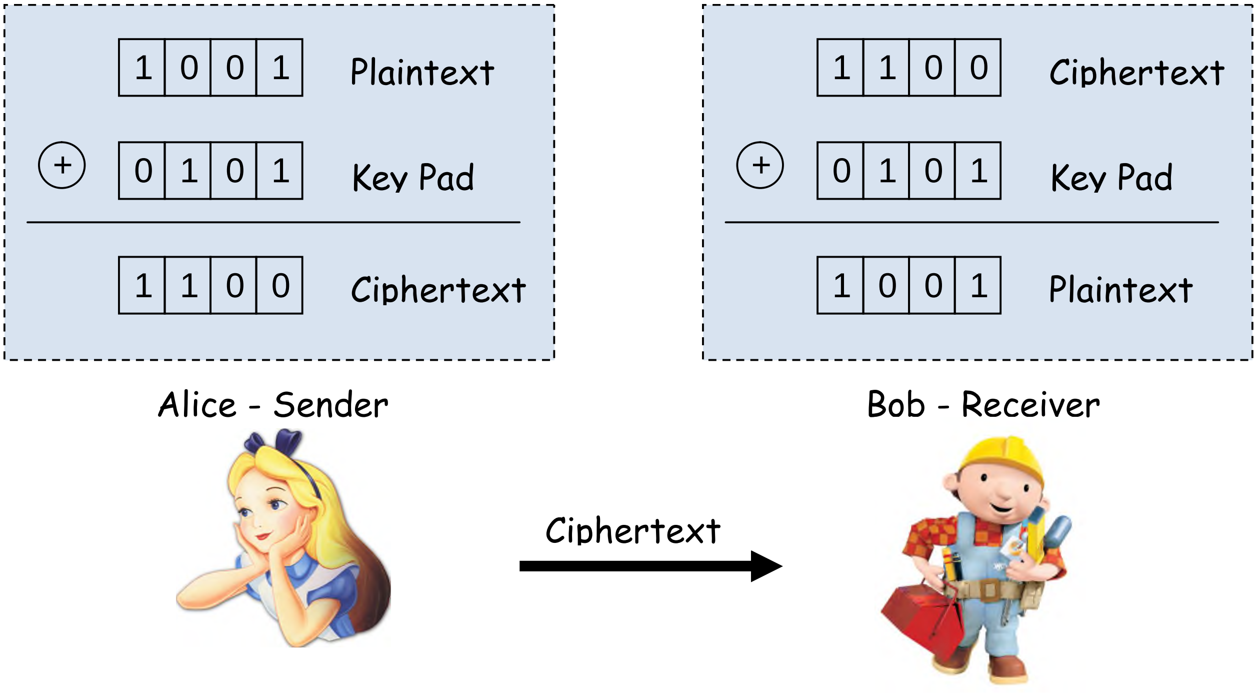


Figure 2 – The One-Time Pad

For the OTP to be secure, the key pad must be 1) truly random, and 2) never reused. Not meeting either of these conditions significantly reduces the strength of the security of the OTP. This lesson was learned by the Soviet Union who during WWII reused one-time pads after distributing them to Soviet intelligence field agents (Benson, 2006). As a result, US and UK intelligence agencies working on Project VERONA were able to easily decode their messages (USGPO, 1997).

To take advantage of the OTP, we must generate and distribute random[4] secret keys to both Alice and Bob equal in length to the sum of the lengths of all messages to be

[4] The generated key must be truly random. This is not a trivial requirement but due to space constraints we will not expand on this point.

exchanged. In practice, this places a significant burden on key distribution and management as one must continuously generate and distribute key pads between the authorized entities in a secure manner. For this reason, historically the OTP is only used in environments which justify the costs involved with secure key distribution. However, QKD offers an attractive point-to-point solution to this dilemma.

Quantum Key Distribution (QKD)

The beginning of QKD can be traced back to Stephen Wiesner who developed the idea of quantum conjugate coding in the late 1960's (Wiesner, 1983). As a student at Columbia University, he described two applications for quantum coding: a method for the creation of fraud-proof banking notes (quantum money) and a method for the transmission of multiple messages in such a way that reading one of the messages destroys the others (quantum multiplexing). Wiesners's quantum multiplexing utilizes photons polarized in conjugate bases as quantum bits (qbits) in order to pass information. In this manner, if the receiver measures the photons in the correct polarization basis, they will receive a correct result with high probability. However, if the receiver measures the photons in the incorrect (conjugate) basis, they will obtain a random result and destroy all information about the original basis. In 1984, Charles Bennett and Gilles Brassard exploited this concept when they proposed the first QKD protocol, BB84, for secure communication (Bennett and Brassard, 1984).

The BB84 Protocol

In the BB84 protocol, the qubits are single photons polarized into one of four polarization states selected from one of two conjugate basis sets as shown in Figure 3. The Rectilinear Basis is shown in yellow and is comprised of 0° and 90° polarizations, and the Diagonal Basis is shown in blue and is composed of 45° and 135° polarizations.

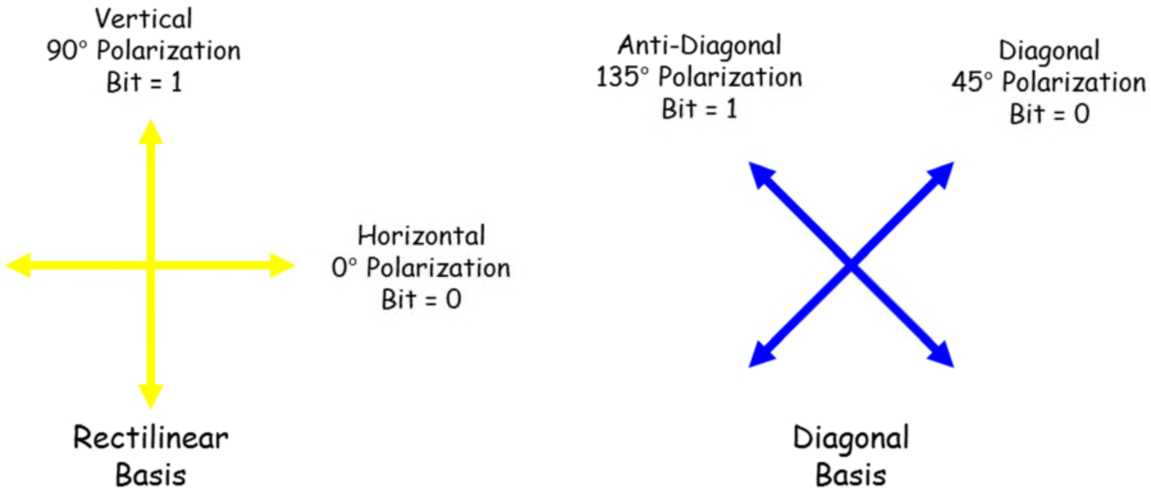


Figure 3 – The Rectilinear and Diagonal Bases Sets

Alice randomly generates both a bit (0 or 1) and a basis (Rectilinear or Diagonal), polarizes a photon accordingly, and sends it to Bob. Bob randomly selects a basis (Rectilinear or Diagonal) to measure the arriving polarized photon. If Bob's randomly selected basis matches Alice's basis, Bob will measure the bit encoded by Alice with 100% accuracy[5]. If Bob's basis does not match Alice's basis, he will measure the correct bit with 50% probability because a photon polarized in one basis yields a random value when measured in its conjugate basis. By encoding classical bits in two conjugate bases, the BB84 protocol provides a means for passing keys between two parties in such a way that an eavesdropper would destroy information and be detected.

The BB84 protocol consists of a sender, Alice, and a receiver, Bob, connected by a classical public channel and a quantum channel as shown in Figure 4. The classical channel is a conventional authenticated public communications link and is used to conduct error reconciliation during the key exchange process. It is assumed that only passive eavesdropping may take place on the classical channel as long as some initial key material is shared for authentication. The quantum channel is used to exchange the quantum encoded qubits. BB84 is often called a "Prepare and Measure" QKD protocol because Alice prepares the photons and Bob measures them. The BB84 protocol can be broken into four steps: Quantum Exchange, Sifting, Information Reconciliation, and Privacy Amplification.

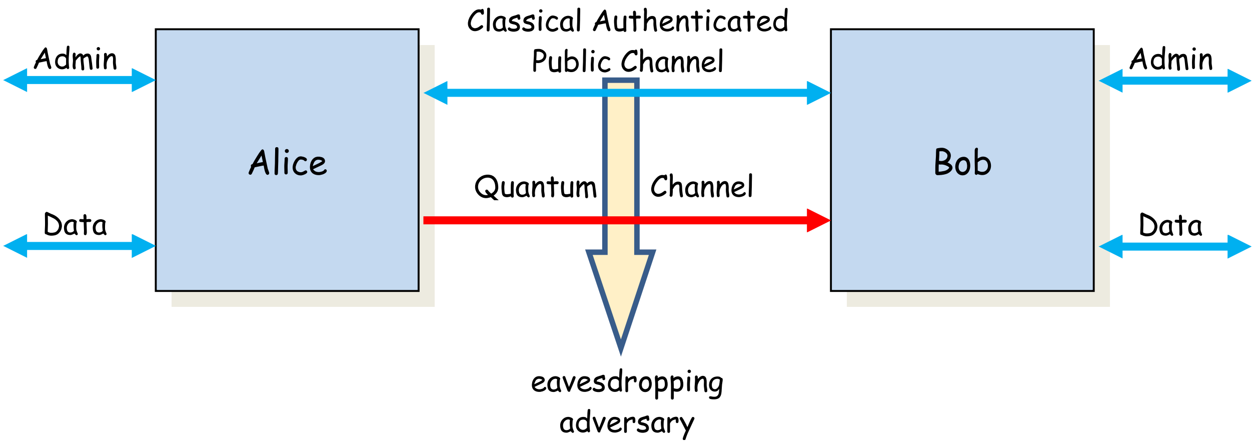


Figure 4 – A BB84 QKD System Block Diagram

Quantum Exchange

The first step in the BB84 protocol is for Alice to generate a random bit string which is the initial candidate key string. She also generates a random bit string which is used to select the basis (Rectilinear or Diagonal) used to encode the bits as polarized photons. Alice transmits one polarized photon to Bob for each bit of the key string using the

[5] In reality, errors can be induced by the environmental noise, equipment non-idealities, or a malicious adversary snooping on the quantum channel.

associated random basis. Bob receives these qubits and randomly chooses a basis in which to measure them. If Bob chooses correctly, he will receive the same bit value that Alice transmitted (assuming perfect transmission and no interference). If he chooses incorrectly, then Bob has a 50% probability of obtaining the correct value.

Sifting

After Bob receives all of the qubits measured in randomly generated bases, he communicates his choice of basis for each qubit to Alice. Alice responds to Bob identifying those bit positions for which Bob's basis agreed with her basis. Alice and Bob both discard any bits for which their measurement basis differed. At this point, Alice and Bob should have an identical set of bits which are a subset of the original candidate key string transmitted by Alice.

Information Reconciliation

Information reconciliation is a very important step in the BB84 protocol as this is where the error rate is calculated and will reveal the presence of an eavesdropper. Errors can result from many sources including non-idealities in equipment, environmental noise, and/or an eavesdropper which causes a mismatch between Alice's and Bob's sifted key. In order to meet the guaranteed security requirement, it is assumed that all errors are due to eavesdropping.

Alice and Bob first conduct error estimation by systematically sampling a random subset of bits from the sifted key and compare them over the open channel. If all the bits agree, then it is likely that they have the same version of the key which can be verified with a hash. The bits exposed during the public comparison are discarded from the final key before the hash is applied. If there are errors present in the sifted key, information reconciliation is used to identify and correct disagreement between Alice's and Bob's sifted key. Error reconciliation requires the exchange of information over the classical public channel and as a consequence it "leaks" some information about the sifted key. For this reason, the method used for information reconciliation is a critical design choice. Common information reconciliation protocols include Cascade, Winnow, and Low Density Parity Codes (LDPC).

Privacy Amplification

During the information reconciliation process, an eavesdropper can gain partial information about the sifted key by eavesdropping both on the quantum channel (where they introduce detectable errors) and on the public channel. Privacy Amplification is used to reduce, and effectively eliminate, an eavesdropper's knowledge of the sifted key to an arbitrary small value. This can be achieved by using a universal hash function chosen at random from a publicly known set of such functions. The inputs to the function are the sifted key and the error rate calculated during the information reconciliation process. A final key is produced that is shortened based on how much information an eavesdropper could have gained about the original sifted key.

A simplified example of the BB84 protocol is shown in Figure 5. The figure shows the quantum exchange, sifting, and error estimation steps. The information reconciliation and privacy amplification processes are not shown for simplicity.

Quantum Exchange															
Alice’s random candidate key bits	0	1	1	0	1	1	0	0	1	0	1	1	0	0	1
Alice’s random basis selection	D	R	D	R	R	R	R	R	D	D	R	D	D	D	R
Polarized photons sent by Alice	↗	↑	↘	→	↑	↑	→	→	↘	↗	↑	↘	↗	↗	↑
Bob’s random basis selection	R	D	D	R	R	D	D	R	D	R	D	D	D	D	R
Bob’s measured received bits	0	0	1	0	1	0	1	0	1	1	1	1	0	0	1
Bob’s basis agrees with Alice’s basis?	N	N	Y	Y	Y	N	N	Y	Y	N	N	Y	Y	Y	Y
Public Discussion															
Bob reports bases of received bits	R	D	D	R	R	D	D	R	D	R	D	D	D	D	R
Alice says which bases were correct			✓	✓	✓			✓	✓			✓	✓	✓	✓
Sifted key			1	0	1			0	1			1	0	0	1
Bob reveals some key bits at random					1									0	
Alice confirms them					✓									✓	
Outcome															
Remaining shared secret bits			1	0				0	1			1	0		1

Figure 5. BB84 Protocol Example (R=Rectilinear, D=Diagonal)

If an eavesdropper present, even with infinite computational power, the best they could do in an ideal system would be to intercept transmissions from Alice, randomly select a basis for measurement, and retransmit those qubits in the basis that they selected. Since an eavesdropper does not know the basis Alice transmitted in, they would be correct, on average, only 50% of the time. When they retransmit the qubits to Bob he will also select a random basis for measurement, and will be correct, on average, 50% of the time. The combination of these steps will introduce a 25% error into the sifted key. Therefore, if Alice and Bob detect an error rate of 25% or higher in their sifted key, they conclude that there is an eavesdropper present and abandon the key exchange process. This is the basis for the unconditional security that the BB84 QKD protocol achieves because Alice and Bob can always detect the presence of an eavesdropper.

Real World QKD Limitations

If it sounds too good to be true, then it probably is. The ideal BB84 protocol assumptions include: 1) Alice emits perfect single photons; 2) the channel between Alice and Bob is noisy but lossless; 3) Bob has single photon detectors with perfect efficiency; and 4) the basis alignment between Alice and Bob is perfect. If these conditions are met, QKD provides for an “unconditionally secure” key exchange, as shown in several mathematical proofs (Mayers, 2001; Renner, Gisin, & Kraus, 2005; Shor & Preskill, 2000). However, many of these assumptions are not valid when building real world systems. For example, the protocol relies on the transmission of single photons because if there are multiple photons sent an adversary may be able to intercept and measure a photon while letting the remaining photons pass unaffected. Reliable on-demand photon generation is not currently practical, so instead a weak coherent photon pulse is generated and attenuated so that on average there are only 0.1 photons in each packet (that means only 1 in 10 packets will contain a photon). This significantly reduces the efficiency of the protocol, but is required to limit and bound the knowledge gained by an eavesdropper. At the receiving side, single photons must be reliably detected. Unfortunately, single photon detectors are rate limited, have low detection efficiencies, and spuriously trigger. Even when there is no eavesdropper present, the physical characteristics of the quantum channel can introduce errors which affect the polarization of the photons while in transit. The result of these technical limitations is that the final key rate is reduced and errors are introduced into the sifted key even though there is no malicious eavesdropper present. These errors must be corrected before using a cryptographic algorithm since Alice and Bob must have identical copies of the key.

QKD Vulnerabilities

After the release of the BB84 protocol and the realization that an "unconditionally secure" key exchange could exist, several researchers published papers on various ways to attack the protocol, creating the new field of "quantum hacking" (Lütkenhaus, 2000; Myers, Wu, & Pearson, 2004; Scarani, Acin, Ribordy, & Gisin, 2004). Even the creators of the BB84 protocol mused on ways to gain information about the shared key and how to mitigate the leaked information (Bennett, Brassard, Crépeau, & Maurer, 1995). Others have proposed new protocols that minimize the problems with the BB84 protocol, but introduce other problems or require hardware devices with the same limitations as the BB84 protocol (Barrett, Hardy, & Kent, 2005; Bruß, 1998; Cerf, Bourennane, Karlsson, & Gisin, 2002; Grosshans & Grangier, 2002).

Conclusions

QKD provides significant advantages when compared to conventional key distribution. First, the security of QKD security rests on the foundations of quantum mechanics. This is in stark contrast to traditional key distribution protocols which rely on computational security where the computational difficulty of certain mathematical functions is the foundation of security. Second, when using QKD, one can determine if an adversary is eavesdropping on the link because it will induce errors in the key exchange process. In contrast, traditional key exchange algorithms cannot provide any indication of eavesdropping or guarantee of key security.

As with any new technology, there is a gap between the theory and the implementation of the BB84 protocol in the real world. Over the last 28 years, research in the QKD area has matured the technology and resulted in commercial QKD implementations. With the current generation of hardware, QKD systems can only reliably generate share keys over distances of approximately 100km using terrestrial optical fiber systems. As the distance increases, the generated key rate drops. At long distances, QKD systems cannot generate enough key material to support bulk encryption using the OTP. However, commercial QKD systems such as the id Quantique Cerberis[6] system combine a conventional high-speed layer 2 encryption engine with the unconditional security of QKD technology. In

[6] http://www.idquantique.com/network-encryption/cerberis-layer2-encryption-and-qkd.html

this case, the key generated by the QKD system is used as the symmetric key for an Advanced Encryption Standard (AES) bulk encryptor. In this mode of operation, the AES key can be changed based upon the key generation rate of the QKD system. For example, the AES key could be changed once per minute if the QKD system is able to generate at least 128 key bits per minute.

Finally, interest in QKD research continues to grow each year. The race is on to improve the quality of emitters, detectors, and fiber to enable QKD to operate over greater distances and at higher key rates. It is only a matter of time before you will encounter a QKD system in your security infrastructure.

Disclaimer
The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

Bellovin, Steven M. (2011). "Frank Miller: Inventor of the One-Time Pad," Technical Report CUCS-009-11, Department of Computer Science, Columbia University, March 2011. A revised version appeared in Cryptologia 35(3), July 2011. Retrieved 12 March 2012 from https://www.cs.columbia.edu/~smb/talks/crypthistory-otp.pdf

Barrett, J., Hardy, L., & Kent, A. (2005). No signaling and quantum key distribution. Physical Review Letters, 95(1), 10503-10503-10506.
doi:10.1103/PhysRevLett.95.010503

Bennett, C. H. and Brassard, G. (1984), "Quantum Cryptography: Public key distribution and coin tossing", in Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing, Bangalore, p. 175, Retrieved 14 April 2012 from http://www.cs.ucsb.edu/~chong/290N-W06/BB84.pdf.

Bennett, C. H., Brassard, G., Crépeau, C., & Maurer, U. M. (1995). Generalized privacy amplification. Information Theory, IEEE Transactions on, 41(6), 1915-1923.

Brassard, G. (1993). A Bibliography of Quantum Cryptography. ACM SIGACT News, pp. 16-20.
Benson, Robert L. (2006), "The Venona Story," National Security Agency. Retrieved 14 April 2012 from
http://web.archive.org/web/20060614231955/http://www.nsa.gov/publications/publi00039.cfm.

Bruß, D. (1998). Optimal eavesdropping in quantum cryptography with six states. Physical Review Letters, 81(14), 3018-3021. Retrieved from http://arxiv.org/pdf/quant-ph/9805019.pdf

Cerf, N. J., Bourennane, M., Karlsson, A., & Gisin, N. (2002). Security of quantum key distribution using d-level systems. Physical Review Letters, 88(12), 127902. Retrieved from http://arxiv.org/pdf/quant-ph/0107130

Grosshans, F., & Grangier, P. (2002). Reverse reconciliation protocols for quantum cryptography with continuous variables. Paper presented at the Proc. 6th Int. Conf. on Quantum Communications, Measurement, and Computing (QCMC'02), 1-1-5. Retrieved from http://arxiv.org/pdf/quant-ph/0204127.pdf

Hitt, Parker (1916). "Manual for the Solution of Military Ciphers," Press of the Army Service Schools, Fort Leavenworth, Kansas. Retrieved 12 March 2012 from http://openlibrary.org/works/OL92027W/Manual_for_the_solution_of_military_ciphers.

Kahn, David (1996). "The Codebreakers." Macmillan. pp. 397–8. ISBN 0-684-83130-9.

Kerckhoffs, Auguste (1883). "La cryptographie militaire", Journal des sciences militaires, vol. IX, pp. 5–38, Jan. 1883, pp. 161–191, Feb. 1883. Retrieved 12 March 2012 from http://www.petitcolas.net/fabien/kerckhoffs/#english.

Klein, M., "Securing Record Communications: The TSEC/KW-26," National Security Agency, Retrieved 12 March 2012 from http://www.nsa.gov/about/_files/cryptologic_heritage/publications/misc/tsec_kw26.pdf.

Lütkenhaus, N. (2000). Security against individual attacks for realistic quantum key distribution. Physical Review A, 61(5), 052304. Retrieved from http://arxiv.org/pdf/quant-ph/9910093

Mayers, D. (2001). Unconditional security in quantum cryptography. Journal of the ACM (JACM), 48(3), 351-406. Retrieved from http://arxiv.org/pdf/quant-ph/9802025

Myers, J. M., Wu, T. T., & Pearson, D. S. (2004). Entropy estimates for individual attacks on the BB84 protocol for quantum key distribution. Paper presented at the Proceedings of SPIE, , 5436 36-47.

NIST 800-21 (2005). "Guideline for Implementing Cryptography In the Federal Government," Retrieved 12 March 2012 from http://csrc.nist.gov/publications/nistpubs/800-21-1/sp800-21-1_Dec2005.pdf.

Renner, R., Gisin, N., & Kraus, B. (2005). Information-theoretic security proof for quantum-key-distribution protocols. Physical Review A, 72(1), 012332. Retrieved from http://arxiv.org/pdf/quant-ph/0502064

Rivest, Ronald L. (1990). "Cryptology," Chapter 13 of Handbook of Theoretical Computer Science, (ed. J. Van Leeuwen) vol. 1 (Elsevier, 1990), 717-755. Retrieved 12 March 2012 from http://people.csail.mit.edu/rivest/Rivest-Cryptography.pdf.

Scarani, V., Acin, A., Ribordy, G., & Gisin, N. (2004). Quantum cryptography protocols robust against photon number splitting attacks for weak laser pulse implementations. Physical Review Letters, 92(5), 57901. Retrieved from http://arxiv.org/pdf/quant-ph/0211131

Shannon, Claude E. (October 1949). "Communication Theory of Secrecy Systems". Bell System Technical Journal (AT&T) 28 (4): 656–715. Retrieved 12 March 2012 from http://www.alcatel-lucent.com/bstj/vol28-1949/articles/bstj28-4-656.pdf.

Shor, P. W., & Preskill, J. (2000). Simple proof of security of the BB84 quantum key distribution protocol. Physical Review Letters, 85(2), 441-444. doi:10.1103/PhysRevLett.85.441

Tanenbaum, Andy (2008). "Dutch Public Transit Card Broken: RFID replay attack allows free travel in The Netherlands," Retrieved 12 March 2012 from http://www.cs.vu.nl/~ast/ov-chip-card/.

Trappe, W., & Washington, L. C. (2005). Introduction to Cryptography with Coding Theory (2 ed.). Upper Saddle River, NJ: Prentice Hall.

Wootters, W. K., & Zurek, W. H. (1982). A single quantum cannot be cloned. Nature, 299(5886), 802-803.

USGPO (1997), "Commission on Protecting and Reducing Government Secrecy: Brief Account of the American Experience," Report of the Commission on Protecting and Reducing Government Secrecy. VI; Appendix A. US Government Printing Office. Retrieved 14 April 2012 from http://www.gpo.gov/fdsys/pkg/GPO-CDOC-105sdoc2/pdf/GPO-CDOC-105sdoc2-11-1.pdf.

Vernam, G.S. (22 July 1919), "Secret Signaling System," US Patent 1,310,719, Retrieved 12 March 2012 from http://www.google.com/patents?vid=1310719.

Vernam, G.S. (1926)."Cipher printing telegraph systems for secret wire and radio telegraphic communications," Journal of the American Institute of Electrical Engineers, XLV:109–115, February 1926. Retrieved 12 March 2012 from https://www.cs.columbia.edu/~smb/vernam.pdf.

Wiesner, Stephen (1983), "Conjugate Coding," ACM SIGACT News, Winter-Spring 1983, Vol. 15, No. 1, Jan. 1983**.**

4. A Survey of Quantum Key Distribution (QKD) Technologies

Morris, J. D., Grimaila, M. R., Hodson, D. D., Jacques, D., & Baumgartner, G. (2013). A Survey of Quantum Key Distribution (QKD) Technologies. In B. Akhgar, & H. R. Arabnia (Eds.), *Emerging Trends in ICT Security* (1st ed., pp. 141-152). Waltham, MA: Elsevier.

Published in: **Emerging Trends in ICT Security, Chapter 9, Nov. 2013**

12 Pages

CHAPTER

A Survey of Quantum Key Distribution (QKD) Technologies

9

Jeffrey D. Morris[1], Michael R. Grimaila[1], Douglas D. Hodson[1], David Jacques[1], and Gerald Baumgartner[2]

[1]United States Air Force Institute of Technology, Wright-Patterson AFB, OH, USA
[2]Laboratory for Telecommunications Sciences, College Park, MD, USA

INFORMATION IN THIS CHAPTER

- Defining Quantum Key Distribution
- Reviewing Quantum Key Distribution architectures
- Exploring Quantum Key Distribution networks
- Identifying the key technologies for future Quantum Key Distribution research
- Viewing a Quantum Key Distribution usage scenario

Cryptography

Cryptography, the practice and study of techniques for securing communications between two authorized parties in the presence of one or more unauthorized third parties, is the centerpiece of a centuries-old battle between code maker and code breaker [1]. Historically, government and military applications chiefly used cryptography, but today almost everyone is dependent on cryptography to provide security services including confidentiality, integrity, authentication, authorization, and non-repudiation [2].

The strength of commonly used cryptographic algorithms relies on computational security, which means the algorithm is secure if there is a negligible chance of discovering the key in a "reasonable" amount of time using current computational technology [3]. As computational technology progresses, adversaries may be able to acquire enough computational power to decode encrypted messages in a "reasonable" amount of time. In fact, recent developments in quantum computing technology (including supporting algorithms) have placed certain classes of commonly used asymmetric cryptographic algorithms (e.g., those that rely on the difficulty of factoring large numbers into their constituent primes, such as the Rivest, Shamir, and Adleman (RSA) algorithm), at risk [4,1]. The resulting loss of security in commonly used asymmetric public key cryptographic algorithms will likely increase the usage of symmetric cryptographic systems and intensify the need for secure and efficient key distribution.

Emerging Trends in ICT Security. DOI: http://dx.doi.org/10.1016/B978-0-12-411474-6.00009-8
© 2014 Elsevier Inc. All rights reserved.

Quantum key distribution

The genesis of Quantum Key Distribution (QKD) can be traced back to Stephen Wiesner, who developed the idea of quantum conjugate coding in the late 1960s [5]. As a student at Columbia University, he described two applications for quantum coding: a method for creating fraud-proof banking notes (quantum money) and a method for broadcasting multiple messages in such a way that reading one of the messages destroys the others (quantum multiplexing). Wiesner's quantum multiplexing uses photons polarized in conjugate bases as "qubits" to pass information. In this manner, if the receiver measures the photons in the correct polarization basis, he or she receives a correct result with high likelihood. However, if the receiver measures the photons in the wrong (conjugate) basis, the measured result is random, and due to the measurement, all information about the original basis is destroyed.

In 1984, Charles Bennett and Gilles Brassard proposed the first QKD protocol, BB84, for secure key exchange based on Wiesner's ideas [6]. The goal of the system is to provide perfect secrecy during key distribution. Using the BB84 protocol, a sender and receiver "grow" an unconditionally secure secret key by leveraging properties of quantum mechanics in the form of polarized photons that are transmitted from the sender to the receiver. Because of the quantum properties of photons, any operations performed on photons in transit would irrevocably alter their state, which would be detectable by the receiver. Additionally, as stated by the no-cloning theorem, no photon copies can be produced for the purpose of operating on the copies without affecting the original photons.

As in any communications system, errors may be introduced from a wide variety of sources. In the security analysis of QKD systems, all errors are attributed to a hypothetical adversary (Eve) who is attempting to eavesdrop on the key distribution communications. If the errors are below a defined threshold, the two parties involved in the key exchange can distill an unconditionally secure key even in the presence of an adversary. Otherwise, the key exchange is aborted. When a QKD-generated unconditionally secure key is combined with the one-time pad (an unconditionally secure classical symmetric cryptographic algorithm), the result is an unconditionally secure cryptographic system.

Since the BB84 protocol was first proposed, there have been many QKD-related protocols and architectural and technological developments that make implementing a QKD system more practical and commercially viable. In 2001, ID Quantique SA offered and sold the first commercially available QKD system [7]. Today, QKD systems are available globally from sellers in Europe (ID Quantique, http://www.idquantique.com/; SeQureNet, http://www.sequrenet.com/), Australia (Quintessence Labs, http://qlabsusa.com/), North America (MagiQ, http://www.magiqtech.com/MagiQ/Home.html), and Asia (Quantum Communication Technology Co., Ltd., http://www.quantum-info.com).

QKD is suitable for use in any key distribution application that has high security requirements. Existing documented applications include financial transactions and electoral communications [8,9], but there are numerous potential applications in law enforcement, government, and military applications. The commercial systems typically use QKD as a means to produce shared secret keys for use in bulk symmetric encryption algorithms, such as the Advanced Encryption Standard (AES), instead of using the unconditionally secure one-time pad. In this case, the QKD-generated key is used to update the encryption key frequently (e.g., once a minute) greatly reducing the required QKD key generation rate which is inversely related to the distance between the QKD systems. While it is not unconditionally secure, users in the commercial domain consider this an improvement when compared with updating the key less frequently (e.g., daily or monthly).

Quantum key distribution systems

In this section, we present a historical survey of the development of QKD systems and their architectures. We also present research that is focused on developing QKD networks that can broaden the application domains of QKD.

QKD system architectures

The first QKD system: BB84

The first QKD system was a research platform built in 1989 by Bennett and Brassard to produce a physical realization of their QKD theory [10]. This system, built at IBM's Thomas J. Watson research center, was the first system to deal with the issues posed by non-ideal hardware as opposed to the perfect hardware envisioned in QKD theory. The system used a weak-coherent light source to generate light that was focused by a lens, passed through filters, and then polarization modulated using Pockels cells. At the receiver, the entering light first passed through a Pockels cell, selecting the polarization measurement basis, then went to a prism to split the light into two paths leading to photomultiplier tubes that counted the number of photons in each of the two polarization states belonging to the selected measurement basis. The distance between the transmitters and receiver was an air-gap of roughly 30 cm. The platform used the BB84 protocol for key generation and showed that QKD could be implemented using standard, non-ideal components [10].

Los alamos: QKD leaves the laboratory

In 1996, Richard Hughes of Los Alamos Laboratories led a team that built a QKD system using 14 km of underground optical fibers [11]. Hughes' system used the B92 protocol [12] instead of BB84—the B92 protocol requires that the system be able to generate two quantum states rather than four. At the transmitter (Alice), the system used a 1300 nm laser source and an attenuator to produce weak, coherent pulses. These pulses were then directed to a 50:50 fiber coupler that formed the input to an unbalanced Mach-Zehnder Interferometer (MZI) that split the incoming photon packet into two photon packets. In the MZI, the photons that traveled along the long arm of the interferometer were phase modulated relative to the photons that traveled along the short arm of the interferometer. The pulse-to-pulse randomly selected relative phase encoding provided the bit and basis value used for key exchange. At the output of the MZI, the two time-separated photon packets were injected into the 14 km fiber leading to the receiver.

At the receiver (Bob), the two incoming photon packets entered another unbalanced MZI identical to the one at the transmitter. The phase modulator in the long arm of Bob's MZI was used to select the measurement basis. At the output of the MZI, the photon packet that passed through both the long arm of Alice's MZI and the short arm of Bob's MZI interfered with the photon packet that passed through the short arm of Alice's MZI and the long arm of Bob's MZI. The results of this interference were measured using a time-gated single-photon Avalanche Photo Diode (APD) detector.

Plug and play: QKD made easier

In 1996, Muller et al. created a hardware-protocol system, based on Faraday mirrors, between the cities of Nyon and Geneva [13]. This system is notable for automatically compensating for birefringence and polarization-dependent losses in the transmission fiber. This system attached easily to existing telecommunication fibers with no need for adjustment of the QKD systems, leading to the name “plug and play.” This system has heralded the beginning of QKD transitioning from the domain of the physics laboratory to existing infrastructure.

The key to overcoming the effects of birefringence in the “plug and play” architecture is the use of Faraday mirrors. In this architecture, Bob generates two classical level light pulses that he sends to Alice. Alice reflects the two pulses using a Faraday mirror (which rotates the polarization of the incoming light so that the reflected light is orthogonally polarized to the incoming light), phase modulates one of the two pulses relative to the other, and attenuates both pulses to the single-photon level. The fact that the single-photon light pulses returning to Bob have a polarization orthogonal to what they had when they traveled to Alice undoes any birefringence effects the pulses experienced when traveling to Alice.

Damien Stucki and his teams used variations of the “plug and play” architecture to connect Geneva and Lausanne [14], a distance of 67 km, over regular telecom fiber using the BB84 protocol.

First entanglement-based system: EPR and Bell’s theorem

In 2004, a team from the University of Geneva proposed and built a system utilizing Ekert’s E91 QKD protocol based on quantum entanglement and Bell’s theorem. This protocol uses Bohm’s version of the Einstein-Podolsky-Rosen (EPR) experiment and Bell’s theorem to test for eavesdropping [15]. This system was one of the first to demonstrate the use of entanglement in QKD. The team’s goal was to demonstrate a system using violations of Bell’s inequalities as the foundation for secure key exchange.

This system uses time-bin entangled qubits created from a laser pulse sent through an unbalanced Michelson interferometer (short and long leg), then through a type 1 Lithium Triborate (LBO) nonlinear crystal, where spontaneous parametric down conversion creates a pair of entangled photons. The photons pass through the transmission channel to both Alice and Bob, who perform projective measurements using the same type of unbalanced interferometer. Here, a significant difference from other QKD systems is that the photon source is not at either Alice’s or Bob’s location, but at a third location. By placing the emitter between Alice and Bob, the system is able to exchange keys at twice the distance of a conventional system with the same loss.

By placing the photons in separate time-bins with two different phases and using the Clauser-Horne-Shimony-Holt (CHSH) Bell inequality, an upper bound for correlations can be determined. By scanning the phases, one of the CHSH parameters can be inferred and the correlation coefficients of the CHSH inequality determined. These coefficient values prove a violation of the CHSH inequality. Nicholas Gisin proved in 2002 that when the Bell inequality is violated, the entangled photons can be used in QKD [16].

Continuous variable QKD: short-ranged but fast and secure

The first Continuous Variable QKD (CV-QKD) system debuted in the European SEcure COmmunication based on Quantum Cryptography (SECOQC) network, with the prototype built expressly for the project. Timothy Ralph first described the CV-QKD protocol in 1999, with several variations proposed by Cerf, Assche, Lutkenhaus, and Grosshans. These protocols use squeezed Gaussian states of light that have classical intensity levels to carry information, rather than discrete single photon states [17–20].

This system encodes information in the amplitude and phase of the classical light level beam and produces high rates of key generation over a short distance (such as a metropolitan network), as it is not as sensitive to individual photon loss as the discrete-variable protocols. Originally, it was not suitable over long distances because the higher noise ratios in longer fibers created errors in the quadratures of pulses, interfering in the homodyne detection, but improvements in post-processing have increased the transmission range. The protocol is resistant against general and collective eavesdropping attacks, and has a security proof for coherent attacks as well. These security proofs show it is more secure than some other types of QKD systems against attacks that exploit the non-ideal hardware flaws of QKD systems [21].

QKD networks

Photon loss between the transmitter and receiver due to attenuation in the optical fiber connecting the transmitter with the receiver, coupled with dark counts at the receiver's detectors, dramatically limits the maximum effective range of QKD compared to classical optical telecommunications. However, even with these range limitations, researchers have implemented small-scale QKD networks to demonstrate its potential. The next section describes some of these networks.

DARPA network: introducing layers

In 2002, the Defense Advanced Research Projects Agency (DARPA) built a QKD system to explore networks that had multiple Alice and Bob system pairs rather than a single system [22]. The system integrated QKD-based key generation, traditional Ethernet encryption, and key management to secure a virtual private network (VPN) that was compatible with existing telecommunication infrastructure.

This system is notable for demonstrating QKD using existing security technology and introducing the idea of "trusted relays." This relay system extends the range of a QKD network but introduces the concerns of adequately securing the relays.

SECOQC Network: mixing and matching with nodes

From 2004 to 2008, the SECOQC project operated in Europe to design and set up a network of QKD systems to show the uses of QKD [23]. The network consisted of a collection of point-to-point systems including: three plug and play systems by ID Quantique SA; a one-way weak-pulse system from Toshiba Research in the UK; a Coherent One-way System (COW) by GAP Optique-ID Quantique SA-Austrian Institute of Technology (AIT); an entangled photon system from the University of Vienna and the AIT; a continuous variables (CV-QKD) system by Centre National de la Recherche Scientifique (CNRS), THALES Research and Technology and Université Libre de

Bruxelles; and a free-space link by the Ludwig Maximillians University. The average link length was between 20 km and 30 km, and the longest link was 83 km.

The project created a QKD trusted-repeater network, much like a connected graph, where each vertex is a QKD node and the edges are the QKD communication channels. Each link retransmits key material along the link, so the key hops from link to link. Keys move forward using an algorithm secured with QKD key material to the next node, and the process is repeated until the key reaches its destination. This creates a "trusted repeater" system, where each node is secured to prevent tampering and attack. The network stripped each QKD system of its key distillation functions and set each one to access only the quantum channel. This reduced redundancy between the systems and moved the key management to upper layers of the network [23]. This architecture extends the DARPA network in both number of nodes and the key transmission maximum distance.

SwissQuantum network: simplifying QKD integration

The SwissQuantum network connected three nodes, two in the center of Geneva and one at the European Organization for Nuclear Research (CERN), with a maximum length of 17.1 km, using ID Quantique SA id5100 commercial servers, and ran between March, 2009 and January, 2011. The servers used the BB84 and SARG04 protocols to generate a shared secret key for standard Ethernet network encryptors across a 10 Gbps channel. This network introduced the concept of layers to QKD networks [24].

Adding a mediation layer between the QKD layer and the secure layer allows integration of QKD devices into an existing telecommunication network. The focus of the research was on integration of QKD and the simplification of the plug and play architecture. This network extended the ideas of SECOQC by making it easier to add QKD to an existing network without specially adapting the QKD systems.

Tokyo network: a high-speed network

Much like the SECOQC network, a consortium of schools, industry, and government organizations created a trusted-node QKD network in Tokyo in 2010 to explore the use of many different types of QKD working together [25]. The Tokyo network created a secure environment to demonstrate secure television conferencing, secure mobile phones, and stable long-term operation at high speeds.

Nine organizations from the EU and Japan employed multiple links to demonstrate QKD technologies such as several decoy-state BB84 systems, a Differential Phase-Shift (DPS) QKD demonstrator, an entanglement system, and ID Quantique's commercial QKD system [25]. The network consisted of six links, with a maximum link distance of 90 km. The network was designed as a three-layer architecture using trusted nodes: a QKD node, a key management layer, and a communication layer. The key management layer is notable for passing secret keys between nodes that do not have a quantum channel by using the unconditionally secure one-time pad cryptographic algorithm.

The future of QKD

The short-term future of QKD lies in creating and extending quantum networks. Until the maximum fiber range of QKD hardware increases significantly, long-range QKD communications depends on some form of multi-node networks. These types of networks, while increasing the range and usability of QKD, increase the security concerns of the telecommunications provider. Using a trusted node within a network increases the security overhead and increases the possibility of corruption or hijacking of the quantum channel. Two promising areas of research may help to overcome these types of security issues.

Quantum repeaters

A workable long-range QKD network needs a method of passing qubits from one network segment to the next without destroying the quantum particle. Recall, an observer cannot clone the state of a quantum particle (i.e., the no-cloning theorem) with perfect fidelity. The process of trying to clone a quantum state introduces unavoidable noise in the output, so the copied states generated by the cloning process are not identical to the input state. Hence, the quantum information carriers cannot be copied (amplified) as is typical in classical communication systems.

For a quantum "repeater" to work, the device would need some way of storing the quantum state of incoming quantum particles for a brief time so they can be forwarded to another QKD system. Quantum entanglement, quantum teleportation, and quantum memory are potential tools for building a quantum repeater. Quantum entanglement and teleportation would allow a device to receive a quantum particle from a sender, and then entangle that photon with another photon in the device, which will be sent to a receiver [26]. This idea is the basis of the Quantum Repeaters for Long Distance Fibre-Based Quantum Communication program (QuReP) (http://quantumrepeaters.eu), a project funded by Switzerland, Sweden, France, and Germany. This multiyear project started in 2010 and is scheduled to finish in 2013.

Quantum memory

Quantum memory is a technology needed in order to build a practical quantum repeater. Quantum memory allows a quantum state to be stored for some amount of time before it is read and used. Current research centers on leveraging the properties of rare-earth doped crystals as the basis for holding and emitting the quantum state. This memory stores a copy of the input state, destroying the input state during the process, and can output a perfect copy of the stored input state with high efficiency (>90%). During the output process, the quantum state of the memory is changed, losing the information about the state it emitted. Additional research centers on increasing the emission efficiencies and increasing the number of simultaneous stored states [27].

Continued research using spin-wave storage in the rare-earth doped crystals and experiments in matter-matter entanglement has led to Pr:YSO and Eu:YSO rare-earth crystals that absorb a quantum state and emit that state with high certainty over a multi-minute time frame [28,29]. Improvements in high-speed photon detectors, single photon or pure-state emitters, and the interfaces between disparate technologies may allow these memory crystals to realize quantum repeaters within the next decade.

Free-space QKD: satellites

With technology advancing terrestrial QKD, advances in free-space QKD are being made. Bennett and Brassard originally demonstrated free-space QKD in their first device, but only over a gap of

30 cm. Various experiments, including some of the networks discussed earlier in this article, used a free-space component and pushed the boundaries for QKD. Improvements in lasers, optical tracking and emitting systems, and computers and software have allowed the free-space QKD distance to far exceed that of terrestrial QKD [30–34].

Free-space QKD, with maximum distances over 140 km, has been demonstrated by an experiment in the Canary Islands of La Palma and Tenerife in 2007 [33]. This experiment showed that QKD could communicate through an atmosphere path length much longer than that between a ground station and a low earth orbit satellite. The experiment used a variation of the original BB84 protocol and used telescopes for tracking and receiving. This experiment showed that, using free-space QKD, a link from ground to an orbiting satellite could establish a secret key between any two ground stations, easing the key distribution problem and potentially secure ground-to-satellite communications.

Chinese researchers are working with free-space experiments with the intention of creating an Earth-to-satellite QKD link. In 2010, one group demonstrated a QKD system using a ground station and a balloon-based transmitter. They developed the optics and tracking systems necessary to deal with high relative angular motion, random motion of the platforms, and atmospheric turbulence that would be found in a ground-to-satellite system [35]. Furthering this research, another group in China recently reported reflecting a beam of photons off an orbiting German satellite that is covered with retro-reflectors. These reflected enough of the single photons back to a receiving telescope to meet the standards of a QKD channel [36]. Nicolas Gisin wrote in 2010 that he would not be surprised if Chinese researchers are the first to demonstrate a QKD link between Earth and a satellite [26].

Device independent QKD (DI-QKD)

QKD provides a way of increasing communications security, but it relies on several assumptions: (i) Alice and Bob use truly random number generators, (ii) Alice and Bob prepare and measure the quantum states exactly as required by the QKD protocol, (iii) Alice and Bob can accurately bound the information that an eavesdropper gains about the key by all methods, and (iv) Alice and Bob use a privacy amplification algorithm that eliminates all of the eavesdropper information about the final key. A major advance in combating this information leakage to the eavesdropper is a relatively new protocol known as Device-Independent QKD (DI-QKD). This QKD protocol makes no assumptions about the hardware used by Alice, Bob, and Eve and goes so far as to assume that Alice and Bob may have no knowledge about how their hardware works. The only requirements are that Alice and Bob randomly select their measurement basis and Eve cannot influence this random selection or know its results until after she can no longer act on the quantum states, and that Eve does not know the results of Alice's and Bob's measurements [37].

The DI-QKD protocol uses a form of Artur Ekert's 1991 entanglement-based protocol proposed by Acin, Massar, and Pironio and uses CHSH inequalities to provide security [38]. It handles the problem that real-life implementations differ from the ideal design. It also makes testing of components easier and covers the scenario where the quantum devices are not trusted [39]. The protocol has been proven secure against collective attacks as long as there is no leakage of classical information from Alice and Bob [37]. Several protocols and experiments have been suggested to take advantage of DI-QKD, including using heralded qubit amplification, extending the range and key rate of normal QKD [40], and one that is valid against most general attacks and based on any arbitrary Bell inequalities, not just those based on CHSH inequalities [41]. Unfortunately, DI-QKD requires high-efficiency near-perfect detectors and provides relatively low key rates due to the need for the near-perfect detections.

Measurement device independent QKD (MDI-QKD)

Though DI-QKD provides increased security for non-ideal devices, there is still a major flaw in today's implementations: that of the "detector loophole." The "loophole" is that not all entangled photons are detected, there is always loss in a quantum channel, each detector has finite detection efficiency and is potentially susceptible to side-channel attacks. All of these factors disturb the Bell's inequality tests and affect protocols based on such tests, ultimately limiting the key rate and reducing security [40].

A method has been proposed to eliminate all detector side-channel information, thus avoiding the problems with the "detector loophole." Measurement Device Independent QKD (MDI-QKD) portends to double the transmission distance for normal QKD with comparable key rates. MDI-QKD works by assuming that Alice and Bob have near-perfect state preparation of their photons and can send them to an untrusted relay called Charlie, who performs Bell state measurements that project the incoming photons into a Bell state [42]. Note that Charlie can be untrusted or even under the control of Eve.

The MDI-QKD protocol tolerates high losses for communication of up to 200 km, assuming Charlie is placed in the middle. Unlike the DI-QKD, this protocol doesn't require the use of Bell's inequalities and can be used for any QKD system, as long as Alice and Bob have near-perfect state preparation as in phase or polarization-based systems [43].

A military QKD usage scenario

How could the QKD benefit be used in a military environment? Imagine a crisis affecting the United States in the near future. The president enters his command center, receiving information from around the globe carried by satellites and telecommunication circuits. As decisions flow from the center, the secret instructions are carried by regular telecommunications circuits to the Pentagon and have been encrypted by QKD devices providing key material to fast network encryption devices. These devices change their large-bit keys several times a second, making it virtually impossible for cyber adversaries to decrypt the traffic.

Once at the Pentagon, these decisions are coordinated with contingency plans, then orders are generated to forces around the globe. Once the orders and plans are ready, the information is sent to satellites in orbit, again using QKD-secured circuits. Not only are the ground-to-space links hard to intercept, the data is encrypted using the same large-bit network encryptors. The signals are then sent from space to ground stations using the same type of encrypted circuits. On the ground, adversaries may listen in on the space-to-ground communications, but with the encryption key changing so fast, it is impossible to decrypt the data. As the distance for QKD links increases, many more communication circuits could be secured by such systems. The unconditionally secure nature of QKD-generated key material makes it attractive for high security requirements often found in the military domain.

CONCLUSION

QKD provides significant security advantages when compared with conventional key distribution. First, due to the laws of quantum mechanics, an eavesdropper cannot copy the quantum bits used in the key exchange. Second, increases in computing power do not help an adversary, as a QKD-generated key is unconditionally secure. Third, QKD allows the sender and receiver to know if there is an eavesdropper listening in on the key exchange. Fourth, the security of QKD security rests on the foundations of quantum mechanics. As long as an eavesdropper has not discovered new laws of physics, the security premise of QKD holds true. This contrasts with traditional key distribution protocols, which rely on computational security to secure the key [44].

As QKD technology matures, the architecture of systems will change. Since quantum states randomly selected from any two or more non-orthogonal quantum bases can be used to encode information for a QKD system, there are many ways to implement such a system. Current research focuses on new types of QKD, such DI-QKD and MDI-QKD, which provide methods to overcome the security limitations of existing hardware. As interest in QKD continues to grow and commercial QKD systems become more common, so will the research efforts focused upon improving the quality of emitters, detectors, and fiber to enable QKD to perform over greater distances and at higher key rates.

Disclaimer

The views expressed in this chapter are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the US Government.

Acknowledgments

This work was supported by the Laboratory for Telecommunications Sciences [grant number 5713400-301-6448].

References

[1] Singh S. The code book: the secret history of codes and code-breaking. 17th ed. London: Fourth Estate; 1999.

[2] Barker EB, Barker WC, Lee A. NIST special publication 800-21 guideline for implementing cryptography in the federal government NIST. [Internet]. 2005. Retrieved from: <http://csrc.nist.gov/publications/nistpubs/800-21-1/sp800-21-1_Dec2005.pdf>.

[3] Schneier B. In: Sutherland P, editor. Applied cryptography: protocols, algorithms, and source code in C. 18th ed. New York: John Wiley & Sons, Inc.; 1995.

[4] Loepp S, Wootters WK. Protecting information: from classical error correction to quantum cryptography. 1st ed. New York: Cambridge University Press; 2006.

[5] Wiesner S. Conjugate coding. ACM SIGACT News 1983;15(1):78–88.

[6] Bennett CH, Brassard G. Quantum cryptography: Public key distribution and coin tossing. Paper presented at the Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, 175. Retrieved from: <http://www.cs.ucsb.edu/~chong/290N-W06/BB84.pdf>.

[7] ID Quantique SA. Cerberis quantum key distribution (qkd) server. [Internet]. 2011a. Retrieved from: <http://www.idquantique.com/network-encryption/products/cerberisquantum-key-distribution.html>.

[8] ID Quantique SA. Redefining Security Geneva government secure data transfer for elections. [Internet]. 2011b. Retrieved from: < http://www.idquantique.com/images/stories/PDF/cerberis-encryptor/user-case-gva-gov.pdf > .

[9] Weier H. (European quantum key distribution network. [Internet]. 2011. Ludwig-Maximilians-Universität München. European Quantum Key Distribution Network. Retrieved from: <http://edoc.ub.uni-muenchen.de/13320/1/Weier_Henning.pdf>.

[10] Bennett CH, Brassard G, Ekert AK. Quantum cryptography. Sci Am 1992;1:50–7.

[11] Hughes RJ, Luther GG, Morgan GL, Peterson CG, Simmons C. Quantum cryptography over underground optical fibers. Lect Notes Comput Sci 1996;1109:329–42.

[12] Bennett CH. Quantum cryptography using any two nonorthogonal states. Phys Rev Lett 1992;68 (21):3121–4. Retrieved from: <http://www.infoamerica.org/documentos_pdf/bennett1.pdf>.

[13] Muller A, Herzog T, Huttner B, Zbinden H, Gisin N. "Plug and play" systems for quantum cryptography. Appl Phys Lett 1997;70(7):793–5. Retrieved from: <http://arxiv.org/pdf/quant-ph/9611042>.

[14] Stucki D, Gisin N, Guinnard O, Ribordy G, Zbinden H. Quantum key distribution over 67 km with a plug&play system. New J Phys 2002;4:41–8. Retrieved from: <http://arxiv.org/pdf/quantph/0203118>.

[15] Ekert AK. Quantum cryptography based on Bell's theorem. Phys Rev Lett 1991;67(6):661–3.

[16] Gisin N, Ribordy G, Tittel W, Zbinden H. Quantum cryptography. Rev Mod Phys 2002;74(1):145–95. Retrieved from: <http://arxiv.org/pdf/quantph/0101098>.

[17] Cerf NJ, Lévy M, Van Assche G. Quantum distribution of gaussian keys using squeezed states. Phys Review A 2001;63(5): 052311-1. Retrieved from: < http://dx.doi.org/10.1103/PhysRevA.63.052311> .

[18] Grosshans F, Grangier P. Reverse reconciliation protocols for quantum cryptography with continuous variables. [Internet]. 2002. Paper presented at the Proceedings of the Sixth Internationl Conference on Quantum Communications, Measurement, and Computing (QCMC'02). Retrieved from: < http://arxiv.org/pdf/quant-ph/0204127.pdf > .

[19] Grosshans F, Van Assche G, Wenger J, Brouri R, Cerf NJ, Grangier P. Quantum key distribution using gaussian-modulated coherent states. Nature 2003:421. Retrieved from: <http://arxiv.org/pdf/quant-ph/0312016>.

[20] Silberhorn C, Ralph TC, Lütkenhaus N, Leuchs G. Continuous variable quantum cryptography: beating the 3 dB loss limit. Phys Rev Lett 2002;89(16). Retrieved from: < http://dx.doi.org/10.1103/PhysRevLett.89.167901 >

[21] Fossier S, Diamanti E, Debuisschert T, Villing A, Tualle-Brouri R, Grangier P. Field test of a continuous-variable quantum key distribution prototype. New J Phys 2009;11(4):045023-04538. Retrieved from: <http://arxiv.org/pdf/0812.3292>.

[22] Elliott C, Pearson DS, Troxel G. Quantum cryptography in practice. [Internet]. Paper presented at the Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications; 2003; Karlsruhe, Germany. p. 227–238. Retrieved from: <http://arxiv.org/pdf/quant-ph/0307049>.

[23] Peev M, Pacher C, Alléaume R, Barreiro C, Bouda J, Boxleitner W, et al. The SECOQC quantum key distribution network in Vienna. New J Phys 2009;11(7):075001. Retrieved from: <http://stacks.iop.org/1367-2630/11/i = 7/a = 075001>.

[24] Stucki D, Legré M, Buntschu F, Clausen B, Felber N, Gisin N, et al. Long-term performance of the SwissQuantum quantum key distribution network in a field environment. New J Phys 2011;13:1–18. Retrieved from: <http://www.gap-optic.unige.ch/wiki/_media/publications:bib:stucki2011a.pdf>.

[25] Sasaki M, Fujiwara M, Ishizuka H, Klaus W, Wakui K, Takeoka M, et al. Field test of quantum key distribution in the Tokyo QKD network. Opt Express 2011;19(11):10387–409. Retrieved from: <http://arxiv.org/pdf/1103.3566>.

[26] Gisin N, Thew RT. Quantum communication technology. Electron Lett 2010;46(14):965–7. Retrieved from: <http://quantumrepeaters.eu/images/attachments/Gisin10.pdf>.

[27] Afzelius M, Simon C, De Riedmatten H, Gisin N. Multimode quantum memory based on atomic frequency combs. Phys Rev A 2009;79(5):052329. Retrieved from: <http://arxiv.org/pdf/0805.4164.pdf>.

[28] Clausen C, Usmani I, Bussieres F, Sangouard N, Afzelius M, de Riedmatten H, et al. Quantum storage of photonic entanglement in a crystal. Nature 2011;469(7331):508−11. Retrieved from: <http://arxiv.org/pdf/1009.0489.pdf>.

[29] Simon C, De Riedmatten H, Afzelius M, Sangouard N, Zbinden H, Gisin N. Quantum repeaters with photon pair sources and multimode memories. Phys Rev Lett 2007;98(19):190503. Retrieved from: <http://arxiv.org/pdf/quant-ph/0701239.pdf>.

[30] Buttler WT, Hughes RJ, Kwiat PG, Luther GG, Morgan GL, Nordholt JE, et al. Free-space quantum key distribution. ArXiv Preprint Quant-ph/9801006. [Internet]. 1998. Retrieved from <http://arxiv.org/pdf/quant-ph/9801006.pdf>.

[31] Hughes RJ, Nordholt JE, Derkacs D, Peterson CG. Practical free-space quantum key distribution over 10 km in daylight and at night. New J Phys 2002;4(1):43. Retrieved from: <http://arxiv.org/ftp/quant-ph/papers/0206/0206092.pdf>.

[32] Jacobs BC, Franson JD. Quantum cryptography in free space. Opt Lett 1996;21(22):1854−6.

[33] Schmitt-Manderbach T, Weier H, Fürst M, Ursin R, Tiefenbacher F, Scheidl T, et al. Experimental demonstration of free-space decoy-state quantum key distribution over 144 km. Phys Rev Lett 2007;98(1):010504. Retrieved from: <http://www.univie.ac.at/qfp/publications3/pdffiles/2007-02.pdf>.

[34] Weier H, Schmitt-Manderbach T, Regner N, Kurtsiefer C, Weinfurter H. Free space quantum key distribution: towards a real life application. Fortschritte Der Physik 2006;54(8−10):840−5. Retrieved from: <http://arxiv.org/pdf/1204.5330>.

[35] Wang J, Yang B, Liao S, Zhang L, Shen Q, Hu X, et al. Direct and full-scale experimental verifications towards ground-satellite quantum key distribution. Nat Photonics 2013;7(5):387−93.

[36] Yin J, Cao Y, Liu S, Pan G, Wang J, Yang T, et al. Experimental single-photon transmission from satellite to earth. Unpublished manuscript. [Internet]. 2013 [retrieved 2013 July 5]. Retrieved from: <http://arxiv.org/pdf/1306.0672v1.pdf>.

[37] Acin A, Brunner N, Gisin N, Massar S, Pironio S, Scarani V. Device-independent security of quantum cryptography against collective attacks. Phys Rev Lett 2007;98(23):230501. Retrieved from: <http://arxiv.org/pdf/quant-ph/0702152>.

[38] Acin A, Massar S, Pironio S. Efficient quantum key distribution secure against no-signalling eavesdroppers. New J Phys 2006;8(8):126. Retrieved from: <http://arxiv.org/pdf/quant-ph/0605246>.

[39] Pironio S, Acin A, Brunner N, Gisin N, Massar S, Scarani V. Device-independent quantum key distribution secure against collective attacks. New J Phys 2009;11(4):045021. Retrieved from: <http://arxiv.org/pdf/0903.4460.pdf>.

[40] Gisin N, Pironio S, Sangouard N. Proposal for implementing device-independent quantum key distribution based on a heralded qubit amplifier. Phys Rev Lett 2010;105(7): 070501. Retrieved from: <http://arxiv.org/pdf/1003.0635.pdf>

[41] Masanes L, Pironio S, Acín A. Secure device-independent quantum key distribution with causally independent measurement devices. Nat Commun 2011;2:238. Retrieved from: <http://arxiv.org/pdf/1009.1567.pdf>.

[42] Lo H, Curty M, Qi B. Measurement-device-independent quantum key distribution. Phys Rev Lett 2012;108(13):130503. Retrieved from: <http://arxiv.org/pdf/1109.1473.pdf>.

[43] Tamaki K, Lo H, Fung CF, Qi B. Phase encoding schemes for measurement-device-independent quantum key distribution with basis-dependent flaw. Phys Rev A 2012;85(4):042307. Retrieved from: <http://arxiv.org/pdf/1111.3413.pdf>.

[44] Scarani V, Bechmann-Pasquinucci H, Cerf NJ, Dušek M, Lütkenhaus N, Peev M. The security of practical quantum key distribution. Rev Mod Phys 2009;81(3):1301. Retrieved from: <http://arxiv.org/pdf/0802.4155>.

5. Towards the Modeling and Simulation of Quantum Key Distribution Systems

Morris, J. D., Hodson, D. D., Grimaila, M. R., Jacques, D. R., & Baumgartner, G. (2014). Towards the Modeling and Simulation of Quantum Key Distribution Systems. *International Journal of Emerging Technology and Advanced Engineering, 4*(2)

Published in: **International Journal of Emerging Technology and Advanced Engineering, Vol. 4, Issue 2, Mar. 2014**

10 Pages

EXPLORING RESEARCH AND INNOVATIONS

International Journal of Emerging Technology and Advanced Engineering

Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 4, Issue 2, February 2014)

Towards the Modeling and Simulation of Quantum Key Distribution Systems

Jeffrey D. Morris[1], Douglas D. Hodson[2], Michael R. Grimaila[3], David R. Jacques[4], Gerald Baumgartner[5]

[1,2,3,4]*Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA*

[5]*Laboratory for Telecommunication Sciences, College Park, MD 20740 USA*

***Abstract*— Quantum Key Distribution (QKD) is a next-generation security technology that exploits the properties of quantum mechanics to enable two parties to generate an unconditionally secure shared secret key. QKD is novel because its security is based upon the fundamental laws of quantum mechanics and not on computational complexity. QKD systems are composed of multiple interconnected electrical, optical, and electro-optical subsystems and computer-based controllers and can be viewed as a complex system (or system of systems). Currently, there is no single simulation framework that supports a high level systems engineering analysis of QKD system architectures. In this paper, we present an evaluation process that considers end user and software developer requirements for the identification and selection of a software framework suitable for modeling, simulation, and analysis of QKD systems.**

***Keywords*—Requirements analysis, simulation environment, discrete-event simulation, quantum key distribution, systems engineering.**

I. Introduction

Cryptography, the practice and study of techniques for securing communications between two authorized parties in the presence of one or more unauthorized third parties, is the centerpiece of a centuries old battle between code makers and code breakers [1]. The strength of commonly used modern cryptographic algorithms relies on “computational security,” which means the algorithms are considered secure if there is a negligible probability of discovering the key in a “reasonable” amount of time using current computational technology [2]. However, recent developments in quantum computing technology and algorithms threaten to place certain classes of commonly used classical cryptographic algorithms, such as the Rivest, Shamir and Adleman (RSA) algorithm, at risk of being compromised [1,3].

In 1984, Bennett and Brassard proposed the first QKD protocol, BB84, to provide perfect secrecy during key distribution [4]. Using a QKD protocol, a sender and receiver create an unconditionally secure secret key by leveraging the properties of quantum mechanics.

QKD enables two parties to securely “grow” a shared secret key since any third-party eavesdropping on the key exchange would introduce detectable errors. An unconditionally secure cryptosystem can be built by combining a QKD-generated key with the One Time Pad (OTP) symmetric key algorithm.

Just as in the early days of computing, each QKD system, whether commercial or research, is a unique implementation based on the theory and principles of QKD using currently available components, protocols, and technology. Since there are no widely accepted security and performance standards for evaluating QKD systems, each system designer architects their system based on their own views and needs. Because of this, there is a need to model QKD implementations to estimate system level attributes such as security, performance, and other technical attributes.

A fundamental truth about QKD technology is that, because of the limitations of technology, it is impossible to build the ideal system described in theory. For this reason, each QKD implementation is only an approximation of the ideal apparatus described in theory. Therefore, our research effort is focused on the development of a QKD modeling and simulation framework that will allow system implementation non-idealities to be included in the system analysis so their impact on overall system performance and security can be better understood [5].

The need to evaluate different QKD system implementations coupled with the cost of the systems, the cost of testing, the uniqueness of each system implementation, and the relative scarcity of resources creates a problem: How does one design, develop, test, and analyze QKD systems in a resource-constrained environment, where it is impractical and cost prohibitive to design, develop, build, and acquire the systems?

A practical alternative to testing the actual hardware is to develop a simulation capability that can accurately model a wide variety of existing and proposed QKD implementations and generate the analysis needed. This includes the ability to model the many different protocols currently used in the implementation of QKD systems, and also the ability to model proposed protocols.

Such a simulation capability needs to address many “concerns,” including the effects due to the quantum properties of light, optical components, single-photon detectors, and the behaviour of complex, interacting QKD software processes present within a QKD system. Such a simulation must facilitate experimentation so that the studies conducted can generate reasonable estimates for system effectiveness, performance, security, and cost based on their architecture, components, and processes.

In this paper, we present an evaluation methodology for selecting a simulation tool suitable for the development of a QKD modeling, simulation, and analysis framework. The evaluation methodology is based upon best practices and incorporates identified end user and software developer requirements.

II. End User And Developer Requirements

A fundamental requirement for any simulation is the ability to tailor the fidelity (i.e., level of detail) of the modeled system components and processes to satisfy specific requirements. In other words, for a given simulation study, we often need to increase the fidelity of system components critical to the “system under study” while simultaneously reducing the fidelity of components that minimally affect that system. This is done to avoid complicating the analysis with unnecessary detail and reducing the simulation time.

The simulation also needs to be understandable and easily configured by end users, who are typically not computer programmers, yet provide enough flexibility to model and evaluate the many different components and processes of QKD systems. From a software engineering perspective, this implies the development of a common reusable simulation capability (i.e., framework) that includes abstract models to support the development of system components and implementations.

Because QKD is a relatively new technology, changes in hardware, software and processes used to implement a system frequently occur, even within currently available commercial offerings. A robust simulation capability needs to facilitate modeling these changes quickly, efficiently, and without requiring the user to have significant programming expertise.

Finally, everything has an associated cost. While modeling and simulation is much cheaper than the acquisition and testing of real QKD systems, it runs the risk of not having enough resources committed to the project, which can result in a capability inadequate for its task [6].

Given the diversity of QKD implementations, and the desire for a common set of abstractions to support modelling at different levels of fidelity, two clear ‘end’ users can be identified: software developers and system engineering analysts. Requirements focused on accommodating an analyst tend toward the interface ‘friendliness’ and configuration flexibility that selects and exposes model functionality without resorting to “software level” programming concerns. Requirements that are oriented more towards developers include the architectural features of the application itself, abstractions flexible enough to model components at many levels of detail, and inherent capabilities to create ‘friendly’ interfaces for the analyst.

A consultation with Subject Matter Experts (SMEs) in the fields of optical physics and a review of simulation software literature identified four major requirements areas for the end user: technical, usability, flexibility, and Total Cost of Ownership (TCO). Each area has multiple sub-requirements/criteria, many of which were derived using Banks’ advice for evaluating and selecting simulation software [6], the best practices report for the Department of Defense (DOD) Modeling and Coordination office [7], and the [8] European Space Agency (ESA) guide to user requirements [8]. Table I identifies the relevant technical end user requirements, Table II identifies relevant usability end user requirements, Table III identifies relevant flexibility requirements, and Table IV identifies relevant TCO end user requirements.

TABLE I
END USER CAPABILITY REQUIREMENTS (TECHNICAL)

Capability	Notes
Levels of fidelity	Does it support differing levels of model fidelity?
Execution speed	Does it have fast execution (native C++ is used as a fast benchmark)?
General programming languages interface	Does it interface with general programming languages such as C, C++, and Java?
Input flexibility	Does it accept data from external files, databases, spreadsheets, and interactively from the user?
User-built custom objects	Can users build and reuse objects, templates, and submodels?
Model status and statistics	Can it display data at any specified time during the simulation?

TABLE II
END USER CAPABILITY REQUIREMENTS (USABILITY)

Capability	Notes
Statistical analysis	Does it have built-in statistics functions for output?
Graphical interface	Does it have a native graphical interface?
Documentation support	Does the simulation software have user's guides, etc.?
Reports function	Does it have built-in reports that can be changed by the user?
Output export	Can it output to many different file types?
Training available for simulation software	Does the developer or 3rd party provide training?
User community available	Is there a robust and open user community for the software?

TABLE III
END USER CAPABILITY REQUIREMENTS (FLEXIBILITY)

Capability	Notes
Modular capability/composability	Can internal code modules be linked together to form new constructs?
Reconfigurable	Can users change the parameters of a test run?
Time variable	Can it run simulations at varying sim-time speeds?
Portability	Can it be used on different OS/hardware platforms?

TABLE IV
END USER CAPABILITY REQUIREMENTS (TOTAL COST OF OWNERSHIP)

Capability	Notes
Required operating system	Is it free from specific operating system requirements?
Required platform software	Is it free from required pre-installed software packages?
Documentation available	Is there documentation available from the package developer?
Software developer support	Does the developer respond to support requests?
Cost of simulation software	Is it free or have very low procurement costs?
Licensing fees	Is it free or have very low non-commercial licensing fees for use of the software?
Training costs	Is available training free or at very low cost?
Required hardware	Does the software have minimal specific required hardware?

The list of relevant developer requirements is contained in Table V.

TABLE V
DEVELOPER CAPABILITY REQUIREMENTS

Capability	Notes
Framework vs. specific platform	Is it a general development platform rather than a specific topic modeling package?
Interactive internal debugger	Does it have a built-in debugger?
Graphical programmer interface	Does it have a built-in graphical programmer interface?
General programming languages interface	Does it interface with general programming languages such as C, C++, and Java?
Extended library of add-ons	Is there an extended library of add-ons available?
Third-party support	Do 3rd parties support the software?
Specific working environment	Does it need other software for development purposes?
Mature software documentation	Is there mature documentation (written by developer, printed, searchable) available?
Third-party documentation	Is there 3rd party documentation available (books, papers, how-tos, videos)?
Randomness	Does it have a built-in pseudo-random number generator?
Hybrid	Can it handle mixed continuous-time and discrete-time models?

III. SELECTING A SIMULATION SOLUTION

In this section, we briefly review different types of simulation paradigms, the problem sets best addressed by each paradigm, and best practices in modelling and simulation. Several seminal papers reaching back to the 1960s were identified during our literature search [9-13]. A review of the modeling and simulation literature provided guidance on general criteria to consider when selecting a simulation solution [14-23]. These criteria, in conjunction with our QKD domain knowledge, were used for the evaluation of end user and developer requirements.

Identifying and selecting the "type" of simulation was the first step in the selection of a simulation solution. Cassandras, Banks and Fishman all have [24-26] decompositions of simulation systems that provide some guidance in making this decision. Figure 1 shows Cassandras' version of the decomposition [24].

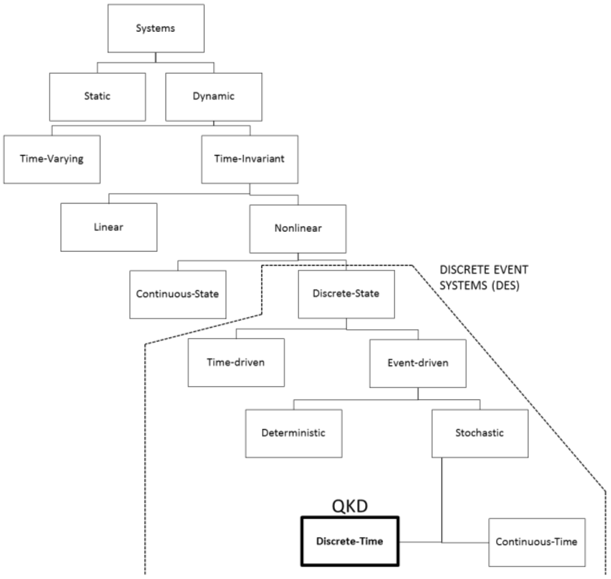


Figure 1. Major simulation type classifications [24].

A major characteristic of the modelling and simulation of QKD systems is that during the simulation of system operation, hundreds of millions of optical pulses will be generated and propagated through the system. While each pulse is most accurately represented as a continuous-time waveform, it is computationally infeasible to model a complete QKD system using continuous-time simulation. This is analogous to the problem of modelling a microprocessor which contains millions of logic gates. While each type of logic gate may be simulated in great detail using continuous-time simulation (e.g., using SPICE [27]), the simulation of a complete microprocessor is conducted using a more abstract, event-based representation (e.g., using Verilog [28]) based upon parameters extracted from more detailed simulations. Similarly, our approach is to model optical pulses as abstract, parameterized objects. In this way, we can manipulate the parameters of the optical pulse when implementing simple transforms such as attenuation or reconstruct the continuous-time optical pulse waveform when necessary in order to implement complex transforms such as interference.

Using the definitions provided by Cassandras with our understanding of the unique properties of the QKD systems we wish to model, we determined the appropriate type of simulation system as follows:

- Static vs. Dynamic – in dynamic systems the current output values depend on past values; all QKD systems use past output data to determine current output values.
- Time-varying vs. Time-invariant – the behavior of time-invariant systems do not change with time. While the behavior of a QKD system does change as a function of time, at the lowest level we choose to treat this temporal behavior using very short discrete time steps during which the system will change very little. Therefore, we treat the desired simulation solution as time-invariant during each discrete time step but allow changes in system behaviour between time steps.
- Linear and Non-linear Systems – in linear systems the output depends linearly on the input, while in nonlinear systems the output is not a linear function of the input. QKD is not a linear process since some of the quantum processes involved, such as measurement, are not linear.
- Continuous-state vs. Discrete-state systems – discrete-state system variables are elements of a discrete set. In contrast, continuous-state variables may take on any state value. When simulating continuous-state QKD functions using a classical computer, these functions are ultimately described by a finite set of numbers. For example, consider the range of possible values of a signed decimal number using a 64-bit binary representation. While the set of possible values is quite large, it is from a countable finite discrete set.
- Time-drive vs. Event-driven – event-driven systems only change when asynchronously generated discrete events occur. The propagation of optical pulses through QKD systems can be described efficiently using a series of scheduled discrete-time events.
- Deterministic vs. Stochastic systems – stochastic systems result when any of the variables are random. QKD is an inherently random process.
- Discrete-time vs. Continuous-time – discrete-time systems have one or more variables that are defined only at discrete points in time, usually the result of some type of sampling process. QKD systems could be described by either system, but a discrete-time system has far less overhead and any continuous-time system can be considered a discrete-time system if the time period is small enough.

Based upon how we intend to model system components, we select a dynamic, time-invariant, non-linear, discrete-state, event-driven, stochastic, discrete-time simulation as the appropriate simulation type. According to Cassandras, this type of simulation system is classified as a Discrete-Event System (DES) as shown in Figure 1.

Selecting a simulation solution that supports the DES modeling paradigm is the first discriminator to narrow simulation solutions under consideration. Another primary discriminator in selecting a simulation solution is the cost of the solution, which is a key end user requirement. While commercial tools may offer advanced modelling capabilities and toolkits, they also come with associated expenses. As a consequence, we focused on candidate DES solutions that were either considered free or open-source. For comparison, we did include commercial tools that are commonly used in mathematically intensive system modeling endeavours in the study.

The initial selection of candidate simulation solutions was driven by a review of available packages identified through Internet-based searches. After reviewing multiple sources that listed or evaluated DES software, a short list of such software became the open-source pool of candidates. We used the following criteria to justify further evaluation of the open-source candidates:

- Still in use (actively developed)
- Source code availability
- Free or low cost

The first criteria ensured that we avoided abandoned software projects without any obvious support or continued development. The second and third criteria came from discussion with SMEs and the literature review. Source code availability enables more developmental freedom in creating a simulation capability suited to evaluate QKD systems. The 'free or low-cost' requirement lowers the cost to end users, a major consideration when commercial software packages can cost in the tens of thousands of dollars and may have substantial yearly maintenance fees.

The other half of the pool included several Commercial-Off-The-Shelf (COTS) software products. Considering these products enabled us to evaluate how well the open-source offerings measured up against the commercial products using the selected criteria. The purpose was to see if there were any critical functional capabilities included in COTS products that were not available in free, open source solutions. It also helped to clarify and understand the value offered by commercial vendors and solutions. One could assume that since commercial products have more dedicated paid resources to enhance products than a typical open-source project, the quality and capabilities of products might trump free open source solutions. A possible counter argument to that assumption is that, due to the lack of source code availability, fewer developers can take part in the development process to enhance the product and correct problems (i.e., bugs). The development of Linux strongly supports the counter argument for operating systems.

The COTS products selected for this evaluation included popular packages in the engineering and physical sciences (e.g., MATLAB/Simulink, Mathematica) along with other dedicated simulation packages (e.g., OpNet, Simul8, Arena).

What follows is the list of chosen packages, starting with the low-cost/free packages:

A. Ptolemy II [29]

Ptolemy II is an open-source framework using an actor-oriented philosophy available from the University of California, Berkeley. Each actor is a base unit and communicates through messages. Actors joined together hierarchically build a larger structure known as a model. In each model, there is a component called as director which sets the semantics of the model and executes the algorithm.

B. SimPy [30]

Simulation in Python (SimPy) is a DES framework for the Python programming language. It provides components for processes, customers, messages, and resources. It includes internal variables to monitor and gather statistics and provides random variates. It has data collection, a built-in Graphical User Interface (GUI) and plotting functions and is capable of interfacing with other Python packages to extend its capabilities. A community of developers created and maintain SimPy.

C. C++Sim [31]

C++Sim is a C++ library for the C++ programming language available from Norwich University. It provides a DES framework providing simulation routines, random number generators, queuing algorithms and thread package interfaces. It provides statistics gathering routines, debugging classes, test routines and is compatible with both C++ and Java. The project appears to be migrating from C++Sim to JavaSim.

D. Adevs [32]

Adevs (A Discrete-EVent System simulator) is a C++ library maintained by the Oak Ridge National Laboratory. It is based on the Parallel DEVS and Dynamic DEVS (dynDEVS) formalisms from Zeigler [33]. Adevs links to an external complier (OpenModelica from the Modelica modeling language) and supports Java.

E. OMNeT++ [34]

OMNeT++ is component-based C++ simulation library and framework for building network simulations. OMNeT++ includes an Eclipse-based Integrated Development Environment (IDE) tailored to support OMNeT++-based model development.

It includes extensions for real-time networks, network emulation, other programming languages (C++ and Java) and database integration. It appears that a relatively large community supports OMNeT++ development.

F. Arena [35]

Arena is a COTS DES from Rockwell Automation. It is designed for stochastic processes using statistical distributions through curve fitting. Random sampling allows for event creation and processing over time. Only registered users of the software may access the program documentation, leading to largest degree of uncertainty in the analysis.

G. MATLAB/Simulink [36]

Simulink is an add-on package for MathWork's MATLAB computation language program, necessitating buying both programs. Simulink uses a block diagram style to assemble hierarchal models. It includes a graphical editor, user-customizable block libraries and math-based solvers for dynamic systems. It can use existing MATLAB algorithms and computations and Simulink results can be imported into MATLAB for analysis.

H. Mathematica [37]

Mathematica is a computational language program from Wolfram Research. Much like MATLAB, its primary purpose is to compute mathematical results. While not precisely a DES-based environment, it can be adapted to provide most DES capabilities.

I. Simul8 [38]

Simul8 is from Simul8 Corporation, a company that produces only simulation software. Simul8 provides processing of discrete entities as a tool for planning, design, optimization and engineering. It uses dynamic DES and includes a GUI, statistical tools, and reports functions. It is designed only for the Windows operating environment, but can be used on other platforms using software emulators.

J. OpNet [39]

OpNet is a communication network simulator from Riverbed Technologies. It provides advanced toolsets to model networks but can be adapted to other network-type models. It has a mature set of libraries and add-on tools using object-oriented modeling. It can use DES and hybrid model simulations in parallel and grid-computing environments and can interface with live systems. It has a GUI and an internal debugger.

It is important to note that while there are many more software packages available, the ones listed above were chosen as representative of the field, with a focus on those typically used in an academic setting.

IV. EVALUATING SIMULATION SOLUTIONS

Each criterion is written to supply a binary solution ("Y" or "N") rather than a variable answer (such as a Likert scale). While most criteria are listed in just one section, there are several that appear in multiple sections, indicating their importance to different capabilities (such as documentation and cost criteria). The evaluation process consisted of the following steps:

- Find the software website on the Internet.
- Find available documentation from the developer/company.
- Review 3rd party websites for software information.
- Search for 3rd party literature.
- Read through available information for each package.
- Evaluate each of the criteria based upon the information collected.

The preferred source of information is the software documentation, but in several cases the documentation was not readily available. In this case, 3rd party information and other information from the developer were used to evaluate the software.

After evaluating the criteria, the "Y" and "N" answers were entered into the criteria matrix with the free/low-cost products listed first, then the COTS products. A "Y" answer was a positive result and colored with green. A "N" answer is considered a negative result and colored in red. An unknown result is a "?" and the cell colored yellow if no determination was possible from the available information.

The matrix then totaled the number of "Y", "N", and "?" answers. To account for the "?" values, each software was given a range of "Y" values, from the maximum "Y" to the minimum. The maximum "Y" value is the number of "Y" values for that software plus the number of "?" values. This allows for each unknown to actually be a positive value. The minimum "Y" is the "Y" value minus the "?" value. This is the worst case as each "?" is considered a negative. This produces a range of "Y" values and is shown in the following span plots. The resulting capability evaluation matrices are shown in Tables VI-XI.

TABLE VI
PROJECT CAPABILITY EVALUATION MATRIX

	Ptolemy	SimPy	C++ Sim	Adevs	OMNeT++	Arena	Simulink	Mathematica	Simul8	OpNet
Cost	Y	Y	Y	Y	Y	N	N	N	N	N
Source code availability	Y	Y	Y	Y	Y	N	N	N	N	N
Modular	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Discrete event	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

TABLE VII
END USER CAPABILITY EVALUATION MATRIX (TECHNICAL)

	Ptolemy	SimPy	C++ Sim	adevs	OMNeT++	Arena	Simulink	Mathematica	Simul8	OpNet
Levels of fidelity	?	Y	Y	Y	Y	N	Y	Y	N	Y
Execution speed	N	N	Y	Y	Y	?	Y	Y	?	?
Programming languages interface	Y	Y	Y	Y	Y	?	Y	Y	N	Y
Input flexibility	?	Y	Y	Y	Y	Y	Y	Y	Y	Y
User-built custom objects	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Model status and statistics	Y	Y	?	?	Y	Y	Y	Y	Y	Y

TABLE VIII
END USER CAPABILITY EVALUATION MATRIX (USABILITY)

	Ptolemy	SimPy	C++ Sim	adevs	OMNeT++	Arena	Simulink	Mathematica	Simul8	OpNet
Graphical interface	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Documentation support	Y	Y	N	Y	Y	Y	Y	Y	Y	Y
Reports function	Y	N	N	N	Y	Y	Y	Y	Y	Y
Output export	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
Training available for simulation software	N	N	N	N	N	Y	Y	Y	Y	Y
User community available	N	Y	N	N	Y	Y	Y	Y	N	Y

TABLE IX
END USER CAPABILITY EVALUATION MATRIX (FLEXIBILITY)

	Ptolemy	SimPy	C++ Sim	adevs	OMNeT++	Arena	Simulink	Mathematica	Simul8	OpNet
Reconfigurable	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Time variable	Y	Y	?	?	Y	?	Y	N	Y	Y
Portability	Y	Y	Y	Y	Y	N	Y	Y	N	Y

TABLE X
END USER CAPABILITY EVALUATION MATRIX (TCO)

	Ptolemy	SimPy	C++ Sim	adevs	OMNeT++	Arena	Simulink	Mathematica	Simul8	OpNet
Required platform software	N	N	Y	N	Y	Y	N	Y	?	Y
Documentation available	Y	N	Y	Y	Y	Y	Y	Y	Y	Y
Software developer support	Y	Y	N	N	Y	Y	Y	Y	Y	Y
Cost of simulation software	Y	Y	Y	Y	Y	N	N	N	N	N
Licensing fees	Y	Y	Y	Y	Y	N	N	N	N	N
Training costs	Y	Y	Y	Y	Y	N	N	N	N	Y
Required hardware	Y	Y	Y	Y	Y	N	Y	N	N	N

TABLE XI
DEVELOPER CAPABILITY EVALUATION MATRIX

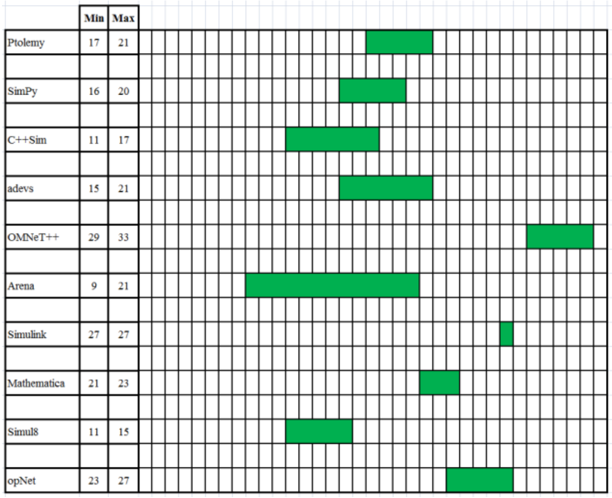
	Ptolemy	SimPy	C++ Sim	adevs	OMNeT++	Arena	Simulink	Mathematica	Simul8	OpNet
Interactive internal debugger	N	N	N	N	Y	?	Y	Y	Y	Y
Graphical programmer interface	Y	?	?	?	Y	Y	Y	Y	Y	Y
General programming languages interface	Y	Y	Y	Y	Y	?	Y	Y	N	Y
Extended library of add-ons	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Third party support	N	N	N	N	?	Y	Y	Y	Y	Y
Specific working environment	Y	Y	N	Y	N	Y	Y	N	Y	N
Mature software documentation	Y	N	N	Y	Y	Y	Y	Y	Y	Y
Third party documentation	N	N	N	N	N	Y	Y	Y	Y	Y
Randomness	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Hybrid	Y	?	Y	Y	?	?	Y	?	Y	?

TABLE XII
OVERALL ADJUSTED SCORE MATRIX

	Ptolemy	SimPy	C++ Sim	Adevs	OMNeT++	Arena	Simulink	Mathematica	Simul8	OpNet
# of "N"	9	10	11	9	3	9	6	8	12	6
# of "Y"	29	28	26	28	35	25	34	31	26	32
# of "?"	2	2	3	3	2	6	0	1	2	2
Total	40	40	40	40	40	40	40	40	40	40
Adjusted N: (N+?)	11	12	14	12	5	15	6	9	14	8
Adjusted Y: (Y+?)	31	30	29	31	37	31	34	32	28	34
max Y ("adjusted Y"-N)	22	20	18	22	34	22	28	24	16	28
min Y (Y-"adjusted N")	18	16	12	16	30	10	28	22	12	24

A span matrix used the criteria scores from Table XII to indicate how various packages compared. The number of "Y" answers from Table XII is on the X-axis, with the software packages on the Y-axis. Each row shows the spread of possible "Y" answers as solid span of color. The larger the number of unknown answers from Table XII, the larger the span for that software.

TABLE XIII
CAPABILITIES ADJUSTED-Y EVALUATION SPAN MATRIX



	Min	Max
Ptolemy	17	21
SimPy	16	20
C++Sim	11	17
adevs	15	21
OMNeT++	29	33
Arena	9	21
Simulink	27	27
Mathematica	21	23
Simul8	11	15
opNet	23	27

V. ANALYSIS OF RESULTS

The results of this evaluation process indicate that the OMNeT++ simulation framework ranked highest/first - it even fared better than commercial offerings. For this initial evaluation or screening process, the result is not surprising. In fact, it can be argued that any screening process such as this is inherently biased (in a good way) towards the solution that best meets the requirements. In other words, the process of establishing the very selection criteria itself helped illuminate some of the driving requirements of special important or of particular concern. For example, the cost of providing an end user the simulation was of particular concern. Because of that, several criteria represent this concern by breaking costs out into different categories such as initial software and licensing fees.

Nevertheless, we consider OMNeT++ a good choice regardless of cost due to other technical concerns related to modeling QKD systems that didn't necessarily show up in the initial screening process – some of these are as follows.

OMNeT's DES provides support to freely intermix the modeling of system features built upon different worldviews and paradigms: For example, we model QKD components and systems using several worldviews including "event-driven" to capture the dynamics of pulse propagation through optical components and fiber cable; "process-based" to capture and represent complex software processes (or process flows) that execute or perform a series of related operations over time (e.g., QKD protocols); and the "object-oriented" that enables us to define standalone software components that can be individually tested, assembled and interconnected to form complete unique system architectures.

OMNeT also provides a substantial amount of well-written documentation and has a vibrant active user community, which were important considerations in our initial selection criteria. For other modeling concerns, the need for sophisticated mathematical capabilities, such as those available in packages such as MATLAB and Mathematica, are simply not required – the standard math functions and capabilities available in almost any programming language was found sufficient to define the effects of interest (e.g., how an optical component affects an optical pulse). Other concerns related to supporting simulations, such as systematically changing input factors, executing replications, collecting data and the subsequent data analysis were found to be sufficiently provided for in OMNeT.

VI. Conclusions

The intent of the evaluation process highlighted in this paper was to select a simulation solution that supports DES modeling concepts with respect to the simulation of QKD systems. A portion of this effort was to identity the necessary requirements for such a simulation and to conduct a search to select the best possible simulation software that meets the requirements.

The research considered different software packages and selected the OMNeT++ open-source software as the best choice for use in the development of a QKD simulation framework. Using a different set of requirements and considerations may result in a different selection.

Further research is in progress and involves creating and documenting a conceptual model for the simulation using modeling formalism. This conceptual model will be the bridge to link the mathematical models provided by the SMEs to the computer code created by the project coders in OMNeT++. The conceptual model is critical in creating well-defined behaviors for the simulation and increasing the validity of the simulation model.

Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

Acknowledgements

This work was supported by the Laboratory for Telecommunications Sciences [grant number 5713400-301-6448].

REFERENCES

[1] Singh, S. (1999). The code book: The secret history of codes and code-breaking (17th ed.). London: Fourth Estate.

[2] Schneier, B. (1995). In Sutherland P. (Ed.), Applied cryptography: Protocols, algorithms, and source code in C (18th ed.). New York: John Wiley & Sons, Inc.

[3] Loepp, S., & Wootters, W. K. (2006). Protecting information: From classical error correction to quantum cryptography (1st ed.). New York: Cambridge University Press.

[4] Bennett, C. H., & Brassard, G. (1984). Quantum cryptography: Public key distribution and coin tossing. Paper presented at the Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, 175, Retrieved from http://www.cs.ucsb.edu/~chong/290N-W06/BB84.pdf.

[5] Scarani, V., Bechmann-Pasquinucci, H., Cerf, N. J., Dušek, M., Lütkenhaus, N., & Peev, M. (2009). The security of practical quantum key distribution. Reviews of Modern Physics, 81(3), 1301. Retrieved from http://arxiv.org/pdf/0802.4155.

[6] Banks, J., Carson, J. S., Nelson, B. L., & Nicol, D. M. (2010). Discrete event system simulation (5th ed.). Englewood Cliffs, NJ: Prentice Hall.

[7] Morse, K. L., Coolahan, J. E., Lutz, B., Horner, N., Vick, S., & Syring, R. (2010). Best practices for the development of models and simulations. (Final Report No. 2010-307). Laurel, MD: John Hopkins University. Retrieved from http://www.msco.mil/documents/10-S-2_26_952%20-%20SIW10F%20-%20MS%20Development%20Best%20Practices%20Final%20Report%20-%20Diem%20-%2020100812%20-%20Dist%20A%20(3).pdf

[8] European Space Agency. (1995). Guide to the user requirements definition phase. (ESA No. PSS-05-02). Noordwijk, The Netherlands: ESA Publications Division.

[9] Basili, V. R., & Weiss, D. M. (1982). A methodology for collecting valid software engineering data. (Technical Report No. TR-1235). Maryland: University of Maryland. Retrieved from http://chess.cs.umd.edu/~basili/publications/technical/T36.pdf.

[10] Cellier, F. E. (1986). Combined continuous/discrete simulation: Applications, techniques and tools. Paper presented at the Proceedings of the 18th Conference on Winter Simulation, 24-33. Retrieved from http://www-oldurls.inf.ethz.ch/personal/fcellier/Pubs/Sim/wsc_86.pdf.

[11] Morris, W. T. (1967). On the art of modeling. Management Science, 13(12), B-707-B-717. Retrieved from http://mansci.journal.informs.org/content/13/12/B-707.full.pdf.

[12] Naylor, T. H., & Finger, J. M. (1967). Verification of computer simulation models. Management Science, 14(2), B-92-B-101. Retrieved from http://mansci.journal.informs.org/content/14/2/B-92.full.pdf.

[13] Shannon, R. E. (1975). Systems simulation: The art and science. Englewood Cliffs, NJ: Prentice-Hall Inc.

[14] Alizai, M., Gao, L., Kempf, & Landsiedel, O. (2010). Tools and modeling approaches for simulating hardware and systems. In K. Wehrle, M. Gunes & J. Gross (Eds.), Modeling and tools for network simulation (pp. 99-99-119). Berlin: Springer Verlag. doi:10.1007/978-3-642-12331-3.

[15] Banks, J., & Gibson, R. R. (1997). Don't simulate when…. IIE Solutions, 9, 30-32. Retrieved from http://www.blueminegroup.com/aai/pdf/10_Rules_Determining.pdf.

[16] Banks, J., & Gibson, R. R. (2001). Simulating in the real world. IIE Solutions, 33(4), 38-40. Retrieved from http://www.blueminegroup.com/aai/pdf/Simulating_RealWorld.pdf.

[17] Banks, J., & Chwif, L. (2010). Warnings about simulation. Journal of Simulation, 5(4), 279-291. Retrieved from http://www.simulate.com.br/warnings.pdf.

[18] Carson, J. S. (1993). Modeling and simulation worldviews. Paper presented at the Proceedings of the 1993 Winter Simulation Conference, Los Angeles, CA. 18-23. Retrieved from http://www.informs-sim.org/wsc93papers/1993_0003.pdf.

[19] Davis, P., & Anderson, R. (2003). Improving the composability of DoD models and simulations. Santa Monica, CA: Rand Corporation. Retrieved from http://www.msco.mil/documents/_3_Improving%20Composability%20of%20DoD%20M&S%20-%2020031113.pdf.

[20] Evans, G. W., Mollaghasemi, M., Russell, E. C., & Biles, W. (1993). Modeling and simulation worldviews. Paper presented at the Proceedings of the 1993 Winter Simulation Conference, 18-23. Retrieved from http://www.informs-sim.org/wsc93papers/1993_0003.pdf.

[21] Geoffrion, A. M. (1987). An introduction to structured modeling. Management Science, 33(5), 547-588. Retrieved from http://mansci.journal.informs.org/content/33/5/547.full.pdf.

[22] Geoffrion, A. M. (1989). Integrated modeling systems. Computer Science in Economics and Management, 2(1), 3-15. Retrieved from http://www.dtic.mil/dtic/tr/fulltext/u2/a215219.pdf.

[23] Gogg, T. J., & Mott, J. R. (1998). Introduction to simulation. Paper presented at the Proceedings of the 1993 Winter Simulation Conference, 9. Retrieved from http://www.informs-sim.org/wsc93papers/1993_0002.pdf.

[24] Cassandras, C. G., & Lafortune, S. (2008). Introduction to discrete event systems (2nd ed.). New York: Springer.

[25] Banks, J. (Ed.). (1998). Handbook of simulation. New York: John Wiley & Sons.

[26] Fishman, G. S. (2001). Discrete-event simulation: Modeling, programming, and analysis. New York: Springer-Verlag.

[27] SPICE. http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/.

[28] Verilog, http://www.verilog.com/IEEEVerilog.html.

[29] Ptolemy, http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.1/ptII/doc/.

[30] SimPy, http://simpy.sourceforge.net/.

[31] C++Sim, http://javasim.codehaus.org/.

[32] Adevs, http://www.ornl.gov/~1qn/adevs/.

[33] Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems (2nd ed.). San Diego: Academic Press.

[34] OMNeT++, http://www.omnetpp.org/.

[35] Arena, http://www.arenasimulation.com/Arena_Home.aspx.

[36] MATLAB/SIMULINK, http://www.mathworks.com/products/simulink/.

[37] Mathematica, http://www.wolfram.com/mathematica/.

[38] Simul8, http://www.simul8.com/.

[39] opNet, http://www.opnet.com/.

6. Reference Architecture for the Analysis of Quantum Key Distribution Systems

Morris, J. D., Mailloux, L. O., Grimaila, M. R., Hodson, D. D., Jacques, D. R., McLaughlin, C., & Holes, J. Reference architecture for the analysis of quantum key distribution systems.

Published in: **Submitted to IEEE Transactions on Emerging Topics in Computing, Aug. 2014**

15 Pages

Reference Architecture for the Analysis of Quantum Key Distribution Systems

J.D. Morris, *Student Member, IEEE*, L.O. Mailloux, *Student Member, IEEE*,
M.R. Grimaila, *Senior Member, IEEE*, D.D. Hodson, D. Jacques, J. Colombi, C. McLaughlin, and J. Holes

Abstract—Quantum Key Distribution (QKD) is an innovative technology which exploits the laws of quantum physics, enabling two parties to generate shared secret cryptographic key. QKD systems offer the unique advantage of being able to detect the presence of an eavesdropper during the key generation process and are suitable for employment in applications requiring high security such as those found in financial, government, and military environments. However, QKD is a relatively nascent technology and real-world systems differ significantly from theory as they are constructed from non-ideal components. The impact of these non-idealities is not well understood due to the complexities of physical and system-level interactions. In this paper, we present a reference architecture to enable the study of polarization modulation-based QKD systems. The reference architecture was developed based on available product specifications, reference literature, and published QKD architectures and is described to provide insight into how QKD systems function. The reference architecture is modeled in a discrete event simulation framework and is used to conduct security and performance analysis to inform design decisions and trade-offs.

Index Terms—Quantum Key Distribution; Architecture; Model & Simulation; System Design; System Analysis

1 Introduction

Quantum Key Distribution (QKD) is the most mature application of quantum cryptography and heralded as a revolutionary technology offering the means for two parties to generate secure cryptographic keying material. Employing the laws of quantum physics, QKD can detect eavesdropping during the key generation process, where unauthorized observation of quantum communication induces discernible errors. However, QKD is a nascent technology where real-world systems are constructed from non-ideal components which adversely impact the security and performance of QKD systems.

In this article, we describe a QKD reference architecture modeled in a discrete event simulation framework used to study the impact of these non-idealities and practical engineering limitations. Moreover, the reference architecture provides a benchmark system to gain additional understanding and study critical design tradeoffs associated with interactions between physical components and system-level considerations (e.g., hardware, software, and protocols). We are focused on bridging the gap between QKD theory and practice; theoretical and experimental physicists are working towards advancing QKD technology, but few are strongly focused on improving the implementation of realized systems. We are using the subject architecture, and derivations thereof, to more fully understand QKD systems.

In Section II, we provide a concise background focused the development of the first QKD protocol, BB84. In Section III, we introduce the discrete event simulation capability used in this work. In Section IV, we describe a polarization-based, prepare-and-measure BB84 QKD reference architecture and describe its constituent elements. In Section V, we present conclusions and future work and an Appendix is presented containing a detailed list of the modeled QKD optical and electro-optical components and their primary behaviors.

2 Background

The genesis of QKD can be traced back to Wiesner, who developed the idea of quantum conjugate coding in the late 1960s [1]. He described two applications for quantum coding: 1) Quantum Money: a method for the creation of fraud-proof banking notes, and 2) Quantum Multiplexing: a method for transmitting multiple messages in such a way that reading one of the messages destroys the other. Charles Bennett and Gilles Brassard subsequently exploited this concept in 1984 when they proposed the first QKD protocol (BB84) and subsequently built their first QKD system in 1989 [2],[3].

Paper submitted August 14, 2014. This work was supported by the Laboratory for Telecommunication Sciences [grant number 5743400-304-6448].

- *J.D. Morris is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: jeffrey.morris@afit.edu).*
- *L.O. Mailloux is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: logan.mailloux@afit.edu).*
- *M.R. Grimaila is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: michael.grimaila@afit.edu).*
- *D.D. Hodson is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: douglas.hodson@afit.edu).*
- *D.R. Jacques is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: david.jacques@afit.edu).*
- *J.R. Colombi is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: john.colombi@afit.edu).*
- *Colin McLaughlin is a Research Physicist at the Naval Research Laboratory, Washington, D.C. 20375 USA (email: colin.mclaughlin@nrl.gov).*

2.1 QKD System Context

In order to provide understanding of how QKD systems are used, we present a QKD system context diagram as shown in Figure 1, illustrating a QKD system consisting of a sender “Alice”, a receiver “Bob”, a classical communications channel (i.e., Internet), and a quantum communications channel (i.e., a dark optical fiber). The QKD system generates a shared secret cryptographic key *K*, which is consumed by Alice and Bob’s bulk encryptors and used to secure a communication link. In the scenario depicted, a plaintext message *m* is transformed into the ciphertext $E_K(m)$ by the bulk encryptors using the key *K*, transmitted over the high-speed data link, and decrypted using the key *K* at the distant end where $m=D_K(E_K(m))$. The bulk encryptors can use any conventional encryption algorithms including DES, 3DES, or AES.

Users desiring increased levels of information protection may use the QKD-generated shared secret key as key material for a One-Time Pad (OTP) encryption algorithm to achieve unconditionally secure communications [4],[5]. However, OTP is not often used due to amount of key required (i.e., equal length to the message) and security requirements (i.e., the key is random and never re-used [5]). QKD systems, when properly implemented, constitute one of the most significant cryptographic developments in recent history.

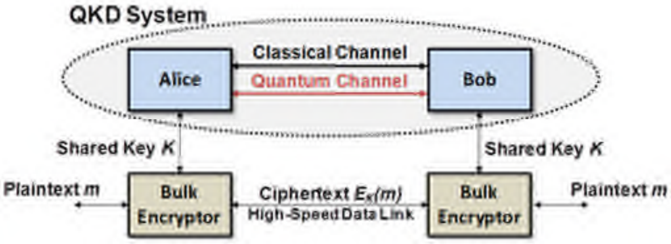


FIGURE 1. QKD System Architecture Context Diagram. Note: additional administrative and control signals are omitted for clarity.

QKD systems primarily use two basic methods for encoding information over the quantum channel [6]. In “prepare-and-measure” protocols, the sender and receiver use matching modulation schemes to encode/decode quantum states using polarization-based or phase-based modulation techniques. In “entanglement” protocols, two or more photons are inextricably linked and are described as a single quantum system, where a measurement on either photon affects the other. In this paper, we focus solely upon the BB84, prepare-and-measure, polarization-based architecture because it was the first QKD protocol, is relatively easy to understand, and remains a popular implementation choice for QKD systems using line-of-sight “free space” direct optical links necessary for future satellite QKD implementations [7].

2.2 BB84 QKD Protocol

In the BB84 protocol, Alice prepares quantum bits (qubits) by encoding information onto single photons using one of two randomly selected conjugate bases (e.g., rectilinear and diagonal) and a randomly selected classical bit value (e.g., 0 or 1) as illustrated in Figure 2. Specifically, Alice encodes each photon using one of the four the polarization states horizontal (oscillating between 0° and 180°), vertical (oscillating between 90° and 270°), diagonal (oscillating between 45° and -135°), or anti-diagonal (oscillating between -45° and 135°). She then transmits the qubit via the quantum channel to Bob, where he measures the photon using a randomly selected basis (e.g., rectilinear or diagonal). Assuming an ideal lossless channel, if Bob measures the qubits in same basis as Alice used, he obtains the encoded bit value with a high degree of accuracy. However, if Bob measures the qubit in the incorrect basis, a random result will be obtained and all previously encoded information is lost. This is due to the quantum-level interactions necessary for measurement to occur, where the mere act of measuring an encoded quantum state collapses the state [8],[9],[10].

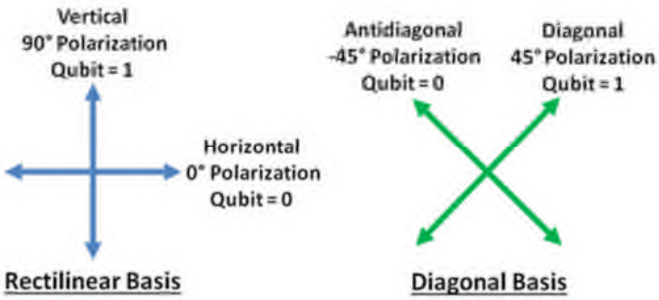


FIGURE 2. Example BB84 Conjugate Polarization Bases: Rectilinear and Diagonal.

Likewise, if an eavesdropper “Eve” attempts to read qubits on the quantum channel, she necessarily introduces detectable errors because she does not know the encoding basis used by Alice a priori. By closely monitoring the error rate of the quantum channel, the Quantum Bit Error Rate (QBER), allows Alice and Bob to determine if an eavesdropper is present during the key generation process. By encoding qubits in two conjugate bases, the BB84 protocol provides a means for securely distributing cryptographic key based on the laws of physics, namely quantum-level uncertainties [8],[9],[10].

2.3 BB84 Idealities

The BB84 protocol assumes several idealities, which are not realistic or practical in real QKD systems, including [10],[11],[12],[13]:

1) On-demand single photon sources in Alice
2) Perfect single photon detection in Bob
3) A lossless quantum channel between Alice and Bob
4) Perfect basis alignment between Alice and Bob

If these conditions are met, QKD provides provably secure key exchange [11],[12],[13]. However, these assumptions are simply not valid when building real-world systems. For example, reliable on-demand single photon generation is not currently practical, commercial single photon detectors have low detection efficiencies, optical fibers have well understood losses, and basis alignment is limited by the accuracy of compensation mechanisms.

To study these non-idealities and similar practical engineering limitations, we developed a specialized simulation framework to enable efficient modeling, simulation, and analysis of QKD system architectures.

3 The QKD Simulation Framework (qkdX)

We have developed a QKD simulation framework (qkdX) to enable efficient modeling of QKD systems, where "fit-for-purpose" QKD system representations can be built for extensive analysis. Practically speaking, this capability allows users (e.g., engineers or analysts) to more quickly model QKD systems, enabling security and performance analysis, including:

1. Model and analyze competing QKD implementations (e.g., variations in hardware components or software processes)
2. Increase understanding of the security-performance design and implementation trade space for realized QKD systems
3. Determine the impact of non-idealities and practical engineering limitations in QKD architectures
4. Identify interactions between physical (quantum phenomenon, temperature, and disturbances) and system-level interactions (hardware designs, software implementations, and protocols)
5. Propose and assess new QKD implementations or protocols
6. Study the security implications of different protocol modifications and system architectures
7. Model and study free-space and space-based QKD systems
8. Maximize research investments and developmental efforts to improve implementations (e.g., should one invest research capital in on-demand single-photon sources or improved single photon detectors?)

3.1 Modeled Components

A list of modeled optical component is shown in Table 1 and described further in the Appendix. Emphasizing the practical nature of our research, we have also chosen to utilize commercially available optical fiber devices and technologies such as telecom wavelength lasers, modulation encoders, and standardized optical components to model QKD systems. Each component is configured with 12-27 adjustable performance parameters and modeled in an event-driven paradigm, while device controllers are modeled in a process-based paradigm. In total, dozens of design decisions and assumptions were made to model these devices and supporting processes based on product specifications provided in the Appendix Reference section, available reference literature, pertinent publications, and discussions with Subject Matter Experts (SME) in the fields of quantum physics, electrical engineering, optical physics, and software engineering.

TABLE 1. Modeled Optical Components.

Attenuator, Fixed Optical	Classical Detector	Laser	Polarization Modulator
Attenuator, Electrical-Variable Optical	Circulator	Optical Switch, 1x2	Single Photon Detector
Bandpass Filter	Half-wave Plate	Polarization Controller	Single-Mode (SM) Fiber
Beamsplitter	In-line Polarizer	Polarization-Maintaining (PM) Fiber	Wave Division Multiplexer
Beamsplitter, Polarizing	Isolator	Polarization Modulator	Faraday Mirror

3.2 qkdX Design Features

Design features of qkdX include: a hybrid discrete-continuous modeling approach to more accurately capture quantum effects; a modular design to allow quick and efficient changes to the system under study; parameterized components allowing for multiple varying instances; and composability allowing for hierarchal construction of complex systems from simple components.

The framework is designed with considerations to support multiple qubit encoding schemes (i.e., polarization-based, phase-based, and entanglement), multiple protocols (e.g., BB84, SARG04, E92), and various QKD applications (e.g., buried optical fiber, terrestrial directional free-space optical link, and multiplexed transmissions).

3.3 qkdX Simulation Considerations

Real-world QKD systems often use existing infrastructure. While the best possible quantum channel would be a dark fiber channel, QKD implementations may time multiplexed weak signal pulses with strong telecommunication signal over the quantum channel. As a consequence, we had to model interference and scattering effects within the optical fiber model. Similarly, as an optical pulse propagates through an optical fiber, it becomes attenuated. In a simulation, it is important to define a threshold below which point the fiber model will delete the optical pulse, otherwise pulses will continue to propagate that are below useful limits.

Another issue requiring special attention is reflections. Each time a pulse enters a component, a small portion is reflected in the opposite direction of travel. These low-power optical packets themselves cause additional reflections. This has the effect of creating a 'reflection storm' of infinite packets bouncing between components in a simulation. To handle this situation, a simulation option was created to enable or disable reflections on a per component basis.

4 The QKD Reference Architecture

The QKD reference architecture serves as a baseline reference for conducting simulation studies. The architecture was developed based on available product specifications, reference literature, and published QKD system designs including [14],[15],[16],[17],[18]. A survey of QKD technologies is provided in [19].

4.1 Alice and Bob Decomposition

Our prototypical QKD system uses a polarization-based, BB84, prepare and measure architecture as shown in Figure 3. The QKD system comprises Alice, Bob, an authenticated public communications channel (classical channel), and a quantum communication channel (quantum channel). Alice and Bob include a system controller, a public channel controller, and a quantum module, respectively. The focus of the reference architecture is the quantum communications path from Alice to Bob, the other components are modeled in a more abstract way. The end-to-end quantum path is discussed in increasing levels of detail throughout the remainder of Section IV.

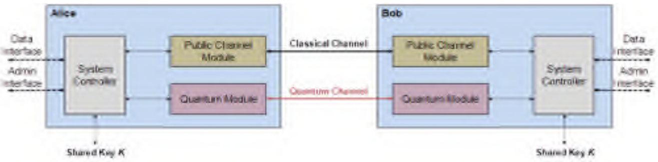


FIGURE 3. QKD System 1st-level Decomposition.

Alice prepares photons with candidate secret key bits and sends them to Bob via the quantum channel. Bob receives the prepared photons and measures them to recover the candidate key bits. Alice and Bob coordinate their system operations by communicating over the authenticated classical channel.

4.2 Alice Subsystem and Quantum Model Decomposition

The Alice subsystem contains several modules including a system controller module, a public channel module, and a quantum module shown in Figure 3. The Alice system controller module is responsible for controlling the Alice subsystem and serves as the master controller to coordinate operations between Alice and Bob. The public channel module interfaces with the system controller module and provides connectivity to the remote system via the classical channel. The quantum module is responsible for generating the quantum state in optical pulses before sending them to Bob via the quantum channel. Alice's quantum module decomposes into nine different subsystems, shown in FIGURE 4Figure 4.

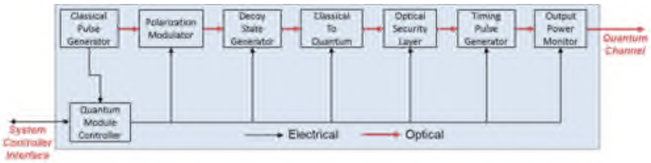


FIGURE 4. Alice Quantum Module Decomposition.

The relative locations of the DSG module and CTQ module are significant for the creation of decoy states within the system. This is a necessary security measure as the DSG creates a 'vacuum' and a 'decoy' state [20] within each frame as a protective measure against the Photon Number Splitting attack [21], a significant security issue for QKD systems. The decoy state has differing levels of photons in relation to the normal 'signal' state but the DSG Electrically-controlled Variable Optical Attenuator (EVOA) applies a large amount of attenuation to create the vacuum state, removing almost all of the photons. Creating these states would be nearly impossible if the pulses travel through the CTQ module and attenuate to quantum levels (generally less than ten photons) before applying the varying values of attenuation to create the vacuum and decoy states.

TABLE 2. Description of Alice Quantum Module Subsystems

Subsystem	Function
Classical Pulse Generator (CPG)	Generates a multi-photon pulse
Polarization Modulator (PM)	Polarizes the photon pulse into the desired polarization
Decoy State Generator (DSG)	Creates decoy states to mitigate photon splitting attacks
Classical to Quantum (CTQ)	Converts classical laser pulses to quantum levels by attenuating to weak-coherent levels
Optical Security Layer (OSL)	Detects optical probing attacks
Timing Pulse Generator (TPG)	Generates a timing pulse used for synchronization and multiplexes signal and timing pulses
Output Power Monitor (OPM)	Monitors the output optical power

The variable attenuators in the DSG and CTQ, both listed in the block diagrams as an EVOA, differ significantly in their usage. The DSG EVOA changes quickly and randomly for each pulse to create the decoy states. The CTQ EVOA attenuates the laser pulses down to quantum level and will only change in response to output levels measured by the OPM module. Their implementation and purpose is very dissimilar, requiring different performance characteristics and requiring the simulation framework to have unique instances of particular component.

4.2.1 Alice Classical Pulse Generator Subsystem Decomposition

The ideal conceptual model of a QKD system specifies polarization-encoded single photons with the desired bit and basis. In reality, reliable on-demand single photon pulse generators are an unrealized technology. Real-world QKD system implementations instead generate a laser pulse containing millions of photons and strongly attenuate the pulse down to statistical sub-photon (quantum) levels. Within the Alice quantum module, the CPG subsystem generates the laser pulses and shifts them into a known polarization. The CPG subsystem contains the components shown in Figure 5.

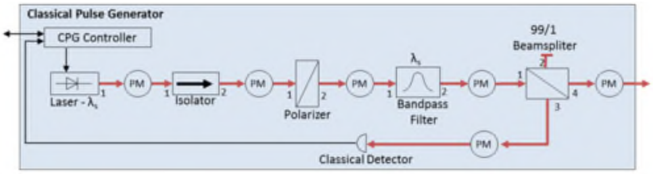


FIGURE 5. Classical Pulse Generator Subsystem Decomposed.

The CPG subsystem components include a controller that has the digital and analog circuits responsible for controlling the laser and monitoring the classical detector. The laser creates optical pulses when it receives a "fire" command from the controller and sends them to the isolator via special polarization-maintaining fiber used in components that cannot have drift in the polarization state. The isolator passes light in the forward direction while significantly attenuating light moving in the opposite direction, assuring that virtually no light (e.g., reflections or light from external sources) enters the laser. The isolator prevents light from entering the laser from downstream component reflections (i.e. the polarizer) and from light entering Alice from the quantum channel. Extraneous light entering the lasing cavity can create perturbations, causing improper output waveforms and disrupting the proper formation of optical packets

The polarizer allows light of one polarization to pass while highly attenuating orthogonal light, this 'cleans' the laser output before sending the light to the bandpass filter. This filter passes optical energy in a narrow band around the laser signal wavelength, λ_S, ensuring only the appropriate signal wavelength leaves the subsystem while preventing other sources of light from entering the laser. The light enters the beamsplitter and is split into two components. The beamsplitter passes 99% of the pulse to the next quantum module subsystem and transmits 1% of the pulse to the classical detector. This detector generates an electrical signal proportional to the power contained in the optical and allows the controller to determine if pulses leaving the CPG have the proper optical power.

4.2.2 Alice Polarization Modulator Subsystem Decomposition

The polarization modulator creates the polarization encoding necessary for the BB84 protocol by polarizing the optical pulses generated by the CPG laser. Some QKD systems use a laser for each polarization value (e.g. four lasers), but this architecture uses one laser and a polarizing component to change each packet polarization. This design decision forced the creation of way to alter the polarization of optical packets, creating a capability that would not be present otherwise. One of the design goals of the simulation framework is to build capabilities into the framework as early as possible to provide 'building blocks' for later architectures. The PM subsystem contains the components shown in Figure 6.

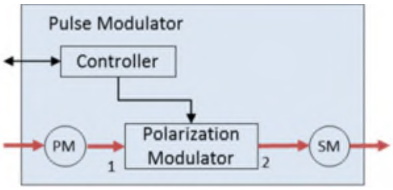


FIGURE 6. Polarization Modulator Subsystem Decomposed.

The PM subsystem includes a controller to interface with the polarization modulator. Unlike the other components in the architecture, the polarization modulator is an abstract component that represents any number of devices used to electronically change the polarization of the light stream from a known polarization to one of several output polarizations. The optical output no longer needs the polarization-maintaining fiber, so the output path changes to single-mode optical fiber. This fiber has a core that guides the light and an outer cladding that reflects the internal light back into the core, allowing for low loss over long distances. Its low loss and significantly cheaper cost than polarization-maintaining fiber make it suitable for telecommunication networks and general fiber optic use.

4.2.3 Alice Decoy State Generator Subsystem Decomposition

The ideal version of a QKD system would emit single photons, but existing hardware is not capable of producing on-demand single photons. This allows eavesdroppers the opportunity to conduct the Photon Number Splitting (PNS) attack, but Alice and Bob have countermeasures in the decoy states. The EVOA inserted into the optical channel allows the quantum controller to randomly vary the power of the optical pulses, allowing creating of the various states (vacuum, decoy and signal). This variance, along with statistics collected on the received states, allows Alice and Bob to detect PNS activity in the quantum channel. Until technology develops a reliable on-demand photon emitter, like the promising research into the quantum dot, some form of defense against the PNS attack is a must for QKD systems. The EVOA contains the components shown in Figure 7.

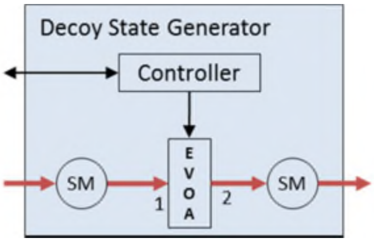


FIGURE 7. Decoy State Generator Decomposed.

The DSG subsystem contains a controller for the EVOA, an opto-electrical device containing a variable attenuator. The attenuator is usually some form of blocking material, such as an opaque slab or a window tilted in the path of the light, connected to an electric motor. As discussed earlier, the selection criteria for this focuses on power attenuation (bleed around the blocking material) and motor speed, as the EVOA needs to quickly change between settings during the quantum exchange. Cost vs. performance will be a major decision for this device. The DSG uses single-mode optical fiber for both input and output into the EVOA.

4.2.4 Alice Classical To Quantum Subsystem Decomposition

Optical pulses generated by the laser in the CPG contain millions of photons, far more than required by QKD protocols. Since single photon generators are not available, existing QKD systems take these classical-level pulses (meaning they contain many photons) and attenuate them to quantum-level pulses (meaning less than ten or so photons) with an average photon count of 0.1. This low number is necessary to achieve the single photon requirements of QKD. Until single photon generators become available, many QKD protocols need both the DSG and the CTQ submodules. The CTQ subsystem contains the components shown in Figure 8.

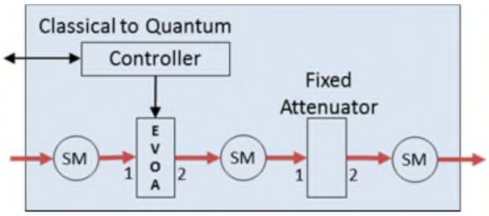


FIGURE 8. Classical To Quantum Decomposed.

The CTQ subsystem contains a controller and an EVOA much like the DSG, but this EVOA does not need the high-speed motors and tight attenuation tolerances. The CTQ EVOA changes position in response to the optical output measured in the final module, so it changes attenuation much less rapidly. The EVOA provides a fine-tuning of the attenuation applied by it and the fixed attenuator to bring the mean photon number of the optical pulses down to single photon levels. The attenuator is usually either doped fibers or misaligned splices designed to apply a fixed attenuation to the optical pulses. All of the interconnections use the standard single-mode fiber, as any polarization changes are corrected at Bob.

4.2.5 Alice Optical Security Layer Subsystem Decomposition

The OSL works to detect and limit the amount of outside light entering the QKD system. The bandpass filter narrows the incoming light frequencies and the circulator routes the incoming light to the classical detector. The detector acts as the 'alarm' by creating an electrical signal to the quantum controller. The circulator allows very little light to pass from port one to three and any that does is heavily attenuated by the isolator. The OSL provides protection against an adversary shining light into the Alice box and discerning information about the internal construction by evaluation the reflections, protects against high-power pulse attacks meant to burn out interior components, and prevents most random frequency light from passing back towards the laser. The OSL subsystem contains the components shown in Figure 9.

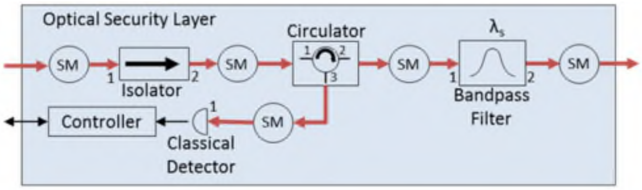


FIGURE 9. Optical Security Layer Decomposed.

Light entering the OSL from outside passes through the bandpass filter, which filters all light around the signal wavelength. Any light managing to pass through filter enters the circulator, which routes light entering it in a clockwise direction, and impacts the classical detector. The detector, usually some form of photo-diode, generates a signal in proportion to the light it receives to the controller, which notifies the quantum controller. In this way, the classical detector functions as an alert device.

Any light that bleeds through the circulator (a very small amount, as the attenuation is greater than 50dB) stops at the isolator, which is oriented to pass light out of Alice. This isolator applies large amounts of attenuation to light travelling towards the laser, effectively stopping light in this direction. Any light travelling from the laser out of Alice passes through the circulator and into the bandpass filter, then out of the OSL, via single-mode fiber interconnections.

4.2.6 Alice Timing Pulse Generator Subsystem Decomposition

The optical frames used by the QKD system need a way to synchronize between Alice and Bob. Even the best timing devices have error and other external timing (i.e. GPS) do not have the accuracy necessary for frame timing. Bob needs to know with a high degree of accuracy when to open the gates windows for his single photon detectors. Alice provides this timing by injecting a bright pulse (i.e. classical-level) of light into the quantum channel to start each frame, followed by a series of quantum-level pulses. Bob uses the bright pulse as a timing 'hack' and opens and closes the 'gates' of his detectors to best detect the quantum pules. Bob uses this timing information when communicating with Alice to during sifting and error correction of the raw key material derived from his detections. The TPG contains the components shown in Figure 10.

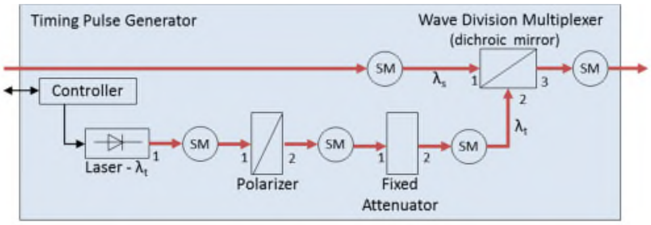


FIGURE 10. Timing Pulse Generator Decomposed.

The TPG subsystem has a controller and laser that work the same as in the CPG, but this laser produces pulses on a timing wavelength slightly different than signal wavelength. The two wavelengths have to be close enough to pass through any bandpass filters, but far enough apart so the two wavelengths can be separated in Bob. The polarizer filters out any extraneous light and the fixed attenuator reduces the pulse power while maintaining classical levels. Finally, the signal and timing pulses are mixed together by the wave division multiplexor, also known as a dichroic mirror, and send via single-mode fiber to the next submodule. In this instance, the Wave Division Multiplexer (WDM) mixes the two signals, but can be used to separate signals, as we see in Bob.

4.2.7 Alice Output Power Monitor Subsystem Decomposition

Alice needs a way to verify her output into the quantum channel is at one or less photons. Pulses with more than one photon are subject to the PNS attack. Components change as they age and some protocols may call for pulses of differing optical power. The OPM allows Alice to sample the outgoing optical packets by using a photon detector capable of counting photons. By sampling the photon numbers, Alice can adjust the CTQ EVOA to output the proper optical power. The OPM contains the components shown in Figure 11.

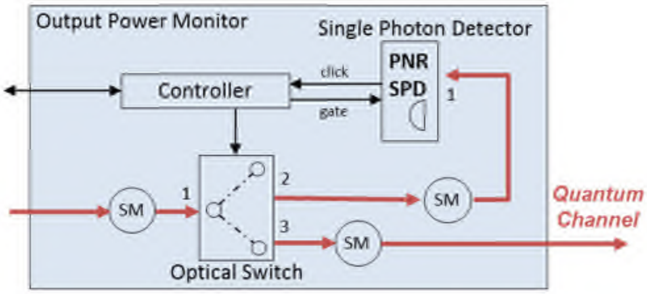


FIGURE 11. Output Power Monitor Decomposed.

The OPM subsystem contains a controller connected to a switch and the photon number resolving single photon detector (PNR-SPD). The optical switch is used to route light between one input port and either out the quantum channel or to the PNR-SPD. The PNR-SPD is an opto-electrical device containing detection equipment and support electronics capable of counting individual photons.

The PNR-SPD in this module is an example of the need to test the framework overriding the knowledge of real systems. Alice would like to know exactly how many photons are entering the quantum channel, and the PNR-SPD gives that count, but these devices are bulky and very expensive. For purposes of the architecture, including this device exercises the framework rather than represent real systems. In real systems, these devices are not present (due to cost and size) and other methods are used to *estimate* the output optical power, increasing the chance that Alice is producing multi-photon packets, enabling the PNS attack. This cost vs. security trade-off makes the inclusion of the DSG (or equivalent) even more important.

4.3 Bob Subsystem Decomposition

The Bob system controller operates the Bob subsystem and receives commands from the Alice controller. Bob's public channel controller works just as Alice's, providing a pathway for authenticated communications over the public channels. Bob's quantum module receives and detects optical pulses over the quantum channel and has no pulse-generating components. It contains timing components that operate its gated detectors using 'bright' pulses in the quantum frames. Bob's quantum module decomposes into five subsystems, shown in Figure 12.

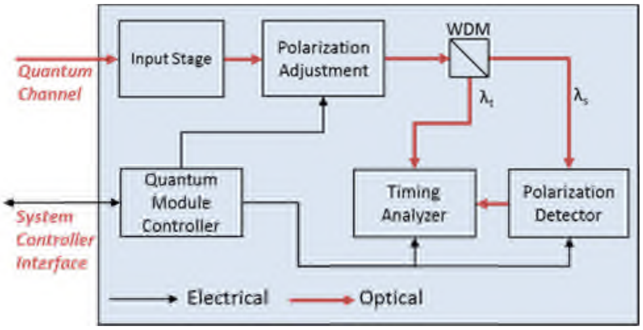


FIGURE 12. Bob's Quantum Module Decomposed.

TABLE 3. Description of Bob Quantum Module Subsystems

Subsystem	Function
Input Stage (IS)	Receives the photon pulse
Polarization Adjustment (PA)	Adjusts for transmission polarization drift
Wave Division Multiplexer (WDM)	De-multiplexes signal and timing pulses
Polarization Detector (PD)	Directs photons to the detectors
Timing Analyzer (TA)	Detects photons during timed gates

Bob's architecture shows many differences from Alice. As Bob's purpose is to receive the few photons that make the journey from Alice, his components are selected to limit the attenuation of the optical signal. Since he is receiving single photons, every bit of attenuation is significant. In Alice, the OSL module is the primary security device against light coming into her system. Bob's Input Stage acts to prevent extraneous light from entering by only admitting a narrow band of frequencies and blocking light entering Bob from reflecting back outside. These components exhibit very small attenuation values, making them a good choice for security while limiting their effects on single photons.

4.3.1 Bob Input Stage Subsystem Decomposition

The Input Stage acts as a filter for incoming light into Bob, as the bandpass filter allows only wavelengths of light around the timing and signal frequencies to enter. The isolator allows light into Bob, but prevents most light (mainly reflections) from leaving Bob. This provides security to Bob by preventing an adversary from using light probes to gain information about internal structure. The Input Stage contains the components shown in Figure 13.

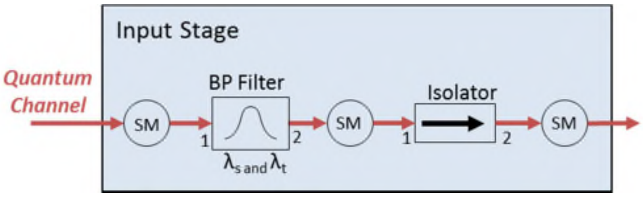


FIGURE 13. Input Stage Decomposed.

4.3.2 Bob Polarization Adjustment System Decomposition

Optical packets undergo changes in polarization as they transit the quantum channel. This polarization drift comes from many factors (i.e., the environment, components, and reflections) so each packet needs correction before entering the detectors. The polarization controller adjusts incoming packets based on the timing signal (bright pulse) received from Alice. The PA subsystem contains the components shown in Figure 14.

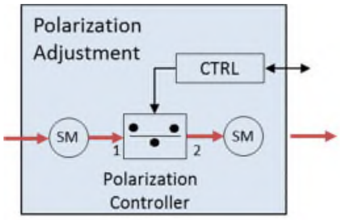


FIGURE 14. Polarization Adjustment Decomposed.

The PA subsystem contains a controller connected to a polarization controller, a device that changes the polarization state of light that passes through it, converting light from any random polarization state into a specific output polarization state. It consists of polarimeter and a state of polarization (SOP) controller combined with a computer and software. The frame of reference polarization between Alice and Bob is set when the system is installed, so Bob always knows what polarization values to expect from Alice.

While this device can change the polarization to any specific point on the Poincare sphere, it has a response time to move from one point to another point. If the change in polarization between successive frames is too great, it will not be able to keep up and the output polarization will not be correct. Such large disturbances are possible if the quantum channel fiber is greatly disturbed, such as aerial fiber being moved by high winds. This response time is another cost vs. performance trade-off, as faster devices cost more than slower devices. The builder would have to take into the account the expected working environment for the QKD system. The still uses single-mode fiber at this point, as the incoming packets should be corrected to the proper polarization.

4.3.3 Bob Wave Division Multiplexer Subsystem Decomposition

This WDM separates the TPG timing signal λ_t, and the CPG signal pulse λ_S into two streams, reversing the mixing done by the WDM in Alice. Once separated, the timing pulses go to the single photon detectors timing controller and the signal pulses output to the polarization detector. The WDM contains the components shown in Figure 15.

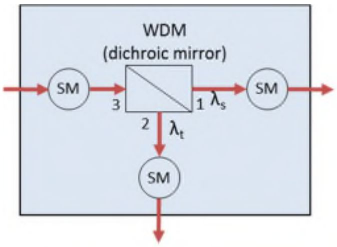


FIGURE 15. Wave Division Multiplexer Decomposed.

This submodule is very simple and exists only to separate the two signals. The choice of the WDM depends on the timing and signal frequencies, which need to be close to one another to pass through the bandpass filters, but far enough apart to be separated by the WDM optics.

4.3.4 Bob Polarization Detector Subsystem Decomposition

The WDM separates the TPG timing signal λ_t, and the CPG signal pulse λ_S. Once separated, the timing pulses go to the timing controller for the single photon detectors and the signal pulses output to the Polarization Detector submodule. The WDM contains the components shown in Figure 16.

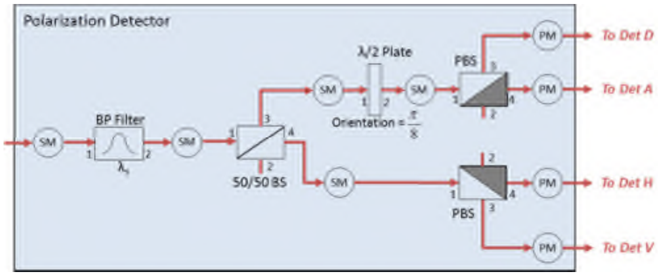


FIGURE 16. Polarization Detector Decomposed.

The PD subsystem contains a bandpass filter which filters the incoming frequencies to clean the signal and a beamsplitter that acts to randomly send single photons to the output ports. One port leads to a polarizing beamsplitter that directs the photon based on its polarization to one of two outputs leading to the Timing Analyzer horizontal and vertical detectors. The other port leads to a half-wave plate that rotates the signal by 45 degrees, and then to a second polarizing beamsplitter that outputs to two separate channels that lead to the antidiagonal and diagonal detectors in the Timing Analyzer. Polarization-maintaining fiber connects the remaining components after the beamsplitters.

4.3.5 Bob Timing Analyzer Subsystem Decomposition

The TA contains single photon detectors set to detect one of four polarizations (antidiagonal, diagonal, horizontal and vertical) and support electronics. The single photon detectors need to be 'gated' open so they are sensitive to single photons but then close to reduce the number of false detections (called dark counts). The classical detector receives the timing signal from the WDM and creates an electrical signal to the timing controller. The timing controller uses the electrical signal from the classical detector to calculate when to open and close the gates of the detectors. The TA contains the components shown in Figure 17.

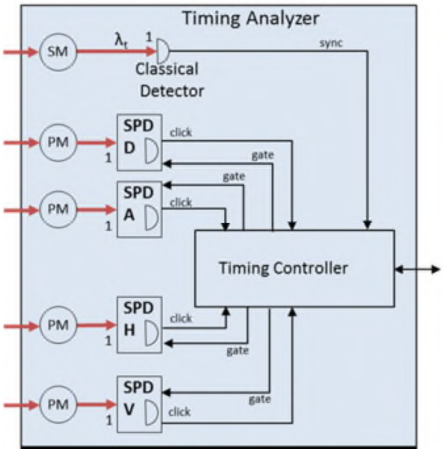


FIGURE 17. Timing Analyzer Decomposed.

The timing controller is a set of electronics that use the electrical signal produced by the classical detector to send commands to the SPDs to open and close the 'gates.' The critical components are the Single Photon Detectors (SPD), which are opto-electrical devices operating in several modes and use signals from the timing controller to change modes. These 'gates' increase the sensitivity of the SPD but increase the chance of a false detection. The four detectors correspond to the four polarization choices from Alice's polarization controller: Diagonal, Antidiagonal, Horizontal, and Vertical.

A design decision for the qkdX framework was to model all optical interference calculations in the SPD, rather than in every component. This choice made the SPD far more complex to model than other components but concentrated the complexity in one spot. The SPD is a critical piece of real QKD systems, as the detector's ability to recover from detections determines the detection rate over any time period, and consequently, the number of photons the SPD can detect. Adding in other considerations for the component means it has many more performance parameters than most components, each one adding to its modeling complexity.

There are several types of SPDs and the selection of what particular SPD to include in the architecture depends on the intent of that particular simulation. Some of the types available include Avalanche Photo Diode (APD), Superconducting Nanowire SPD (SNSPD), and Transition Edge Sensor (TES) with each type has positive and negative attributes. The reference architecture includes a general detector, but the simulation goals determine the correct type.

5 Conclusions and Future Work

In this paper, we have presented a polarization-based, prepare-and-measure BB84 QKD reference architecture used for to test a simulation capability for the efficient modeling and analysis of QKD systems. We provided detailed illustrations of the modeled architecture and accompanying design decisions with detailed discussions of each optical component function and behavior. This paper provides three distinct aspects:

1) Provides an educational tool for understanding QKD architectures
2) Provides detailed discussions of QKD architectural design decisions and trade-offs
3) Serves as a reference "baseline" architecture for conducting security and performance analysis of QKD systems

Future work includes adding the components, controllers, and protocol logic necessary to model other QKD technologies including phase-based and entanglement-based protocols and components, ground-to-air, ground-to-space, space-to-space based systems and the measurement and device independent protocols. Beyond adding new technologies to the simulation frames is work to exercise the existing models, for example, conducting performance studies on decoy states and another on types of SPDs. A forthcoming paper will discuss the results of these studies.

6 Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] S. Wiesner. "Conjugate coding," *ACM SIGACT News* 15(1), pp. 78-88. 1983.

[2] C.H. Bennett and G. Brassard. "Quantum cryptography: Public key distribution and coin tossing," *Proceedings of IEEE International Conference on Computers, Systems and Signal Processing*. 1984, Available: http://www.cs.ucsb.edu/~chong/290N-W06/BB84.pdf.

[3] C.H. Bennett. "Quantum cryptography using any two nonorthogonal states." *Phys. Rev. Lett.* 68(21), pp. 3121-3124. 1992. Available: http://www.infoamerica.org/documentos_pdf/bennett1.pdf.

[4] S.M. Bellovin. "Frank miller: Inventor of the one-time pad," *Cryptologia* 35(3), pp. 203-222. 2011. Available: http://academiccommons.columbia.edu/download/fedora_content/download/ac:135404/CONTENT/cucs-009-11.pdf. C. E.

[5] Shannon. "Communication theory of secrecy systems," *Bell System Technical Journal* 28(4), pp. 656-715. 1949. Available: http://dm.ing.unibs.it/giuzzi/corsi/Support/papers-cryptography/Communication_Theory_of_Secrecy_Systems.pdf.

[6] N. Gisin, G. Ribordy, W. Tittel and H. Zbinden. "Quantum cryptography," *Reviews of Modern Physics* 74(1), pp. 145-195. 2002. Available: http://arxiv.org/pdf/quantph/0101098.

[7] J.-Y. Wang, B. Yang, S.-K. Z. L. Liao, Q. Shen, X.-F. Hu, J.-C. Wu, S.-J. Yang, H. Jiang, Y.-L. Tang, B. Zhong, H. Liang, W.-Y. Liu, Y.-H. Hu, Y.-M. Huang, B. Qi, J.-G. Ren, G.-S. Pan, J. Yin and et. al, "Direct and full-scale experimental verifications towards ground-satellite quantum key distribution," *Nature Photonics*, vol. 7, no. 5, pp. 387-393, 2013.

[8] S. Loepp and W. K. Wooters, Protecting Information, New York: Cambridge University Press, 2006.

[9] M.A. Nielsen and I.L. Chuang, *Quantum Computation and Quantum Information*. Cambridge, UK: Cambridge university press, 2010.

[10] V. Scarani, H. Bechmann-Pasquinucci, N.J. Cerf, M. Dušek, N. Lütkenhaus and M. Peev. "The security of practical quantum key distribution," *Reviews of Modern Physics* 81(3), pp. 1301. 2009. Available: http://arxiv.org/pdf/0802.4155.

[11] R. Renner. "Security of quantum key distribution." 2005. Available: http://arxiv.org/pdf/quantph/0512258.

[12] V. Scarani and C. Kurtsiefer. "The black paper of quantum cryptography: Real implementation problems." ArXiv Preprint arXiv:0906.4547 2009. Available: http://arxiv.org/pdf/0906.4547.pdf.

[13] R. Renner, N. Gisin and B. Kraus. "Information-theoretic security proof for quantum-key-distribution protocols," *Physical Review A* 72(1), pp. 012332. 2005. Available: http://arxiv.org/pdf/quant-ph/0502064.

[14] R. J. Hughes, G. G. Luther, G. L. P. C. G. Morgan and C. Simmons, "Quantum cryptography over underground optical fibers," in *Advances in Cryptology—CRYPTO'96*, Springer Berlin Heidelber, 1996.

[15] A. Muller, T. Herzog, B. Huttner, W. Tittel, H. Zbinden and N. Gisin, ""Plug and play" systems for quantum cryptography," *Applied Physics Letters,* vol. 70, no. 7, pp. 793-795, 1997.

[16] D. Stucki, M. Legre, F. Buntschu, B. Clausen, N. Felber, N. Gisin, L. Henzen and e. al., "Long-term performance of the SwissQuantum quantum key distribution network in a field environment," *New Journal of Physics,* vol. 13, no. 12, p. 123001, 2011.

[17] C. Elliott, "The DARPA quantum network," *Quantum Communications and cryptography,* pp. 83-102, 2006.

[18] M. Peev, C. Pacher, R. Alléaume, C. Barreiro, J. Bouda, W. Boxleitner, T. Debuisschert and e. al., "The SECOQC quantum key distribution network in Vienna," *New Journal of Physics,* vol. 11, no. 7, p. 075001, 2009.

[19] J.D. Morris, M.R. Grimaila, D.D. Hodson, D.Jacques and G. Baumgartner, "A survey of quantum key distribution (qkd) technologies," in *Emerging Trends in ICT Security*, 1st ed., B. Akhgar and H. R. Arabnia, Eds. Waltham, MA: Elsevier, 2013, pp. 141-152.

[20] H.Lo, X.Ma and K.Chen. "Decoy state quantum key distribution," *Phys. Rev. Lett.* 94(23), pp. 230504. 2005. Available: http://arxiv.org/pdf/quant-ph/0411004.

[21] X. Wang. "Beating the photon-number-splitting attack in practical quantum cryptography," *Phys. Rev. Lett.* 94(23), pp. 230503. 2005. Available: http://arxiv.org/pdf/quant-ph/0410075.

OPTICAL COMPONENT APPENDIX

Component Name	**Component Description**
Attenuator, Fixed Optical (FOA)	The Fixed Optical Attenuator is a two port, bidirectional optical component used to reduce the strength of optical pulses as they are transmitted. The amplitude of the output pulse is calculated using the input pulse amplitude, insertion loss, and fixed attenuation of the optical pulse, as shown in the equation below. Note that the fixed attenuation can be changed to values in the range of 0.0 and 80.0 dB. The modeled behaviors are based on [1, 2, 3, 4]. $Amplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-(FixedAttn)}{10}}}$
Attenuator, Electrical Variable Optical (EVOA)	The Electrical Variable Optical Attenuator is a two port, bidirectional optical component used to attenuate optical pulses as they are transmitted. The amplitude of the output pulse is calculated using the input pulse amplitude, insertion loss, and variable attenuation of the optical pulse, as shown in the equation below. Note that the variable attenuation can be set to values in the range of 0.0 and 80.0 dB. The modeled behaviors are based on [5, 6, 3, 7]. $Amplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-(VariableAttn)}{10}}}$
Bandpass Filter	The Bandpass Filter is a two port, bidirectional optical component used to transmit only the desired wavelength of light, while other noise is blocked. The filtering ranges include Central Wavelength (CWL), upper/lower middleband Central Frequency (CF), and upper/lower outer CF. When an optical pulses is transmitted, 1 of 4 situations will occur with regards to the optical pulse's CWL; CWL is the same as the bandpass filter, CWL is outside the bandpass filter's CWL and within the limits of middleband CF, CWL is outside the limits of the middleband CF and within the bandpass filter CF, or CWL is outside the limits of the bandpass filter. The functionality is shown in the equations below, respectively. The modeled behaviors are based on [8, 9, 10, 11, 12]. $E_{output} = E_{input} * \sqrt{10^{\frac{-InsertionLoss}{10}}} * \sqrt{10^{-\left(\frac{bandpass_{CWL}-E_{inputCWL}}{\frac{MidBandWidth}{2}}\right)*\frac{MidBandLoss}{10}}}$ $E_{output} = E_{input} * \sqrt{10^{\frac{-InsertionLoss}{10}}} * \sqrt{10^{\frac{-MidBandLoss}{10}}} * \sqrt{10^{-\left(\frac{MidBandLow_{CF}-E_{inputCWL}}{MidBandLow-OutBandLow}\right)*\frac{(OutBandLoss-MidBandLoss)}{10}}}$ $E_{output} = E_{input} * \sqrt{10^{\frac{-InsertionLoss}{10}}} * \sqrt{10^{\frac{-OutBandLoss}{10}}}$
Beamsplitter	The Beamsplitter is a three port, bidirectional optical component used to split optical pulses into two or more beams. The High Output Percentage (HOP), Low Output Percentage (LOP), and other losses are considered when calculating the amplitude of the output pulse, as shown in the equation below. The modeled behaviors are based on [13, 14, 15, 16]. $E3_{Amplitude} = \sqrt{\left(E3_{x_{output}}\right)^2 + \left(E3_{y_{output}}\right)^2}$ $E3_{x\_output} = E1_{x\_input} * \sqrt{LOP} * \sqrt{10^{\frac{-ExcessLoss}{10}}} * \sqrt{10^{\frac{-PolarDependLoss_x}{10}}} * \sqrt{10^{\frac{-InsertionLoss}{10}}}$ $E3_{y\_output} = E1_{y\_input} * \sqrt{LOP} * \sqrt{10^{\frac{-ExcessLoss}{10}}} * \sqrt{10^{\frac{-PolarDependLoss_y}{10}}} * \sqrt{10^{\frac{-InsertionLoss}{10}}}$
Beamsplitter, Polarizing (PBS)	The Polarizing Beamsplitter is a four port, bidirectional optical component used to split optical pulses in order to separate orthogonal polarizations. This creates orthogonally polarized optical pulses. The amplitude of the output pulse is calculated using the amplitude of the input pulse, extinction power, and other losses, as shown in the equation below. The modeled behaviors are based on [17, 9, 14, 18, 19]. $E4_{ouputAmplitude} = \sqrt{\left(E4_{x\_output}\right)^2 + \left(E4_{y\_extinctionPower}\right)^2}$ $E4_{x\_output} = E1_{x\_input} * \sqrt{10^{\frac{-ExcessLoss}{10}}} * \sqrt{10^{\frac{-PolariztionDependentLosses_x}{10}}} * \sqrt{10^{\frac{-InsertionLoss}{10}}}$

Circulator	The Circulator is a three port, bidirectional optical component used to route optical pulses to the adjacent port. Optical pulses cannot pass into the Circulator in the reverse direction, however an isolate pulse is created if a pulse attempts to pass in reverse direction. The amplitude of the isolate pulse is calculated using the amplitude of the input pulse, insertion loss, and isolation loss, as shown in the first equation below. The amplitude of the forward output pulse is calculated using just the amplitude of the input pulse and insertion loss, as shown in the second equation below. The modeled behaviors are based on [23, 24, 25]. $Amplitude_{Isolated} = Amplitude_{Input} * \sqrt{10^{\frac{-(InsertionLoss+IsolationLoss)}{10}}}$ $Amplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-InsertionLoss}{10}}}$
Detector, Classical	The Classical Detector is a one port, unidirectional optical component used to detect optical pulses passing into the optical receiver port that are strong. Once detected, electrical signals are generated and transmitted through the electrical output port. The amplitude of the output electrical signal is calculated using the conversion factor and peak power of the input optical pulse, as shown in the equation below. Note that optical pulses can only pass into the Classical Detector in the forward direction, not the reverse direction. The modeled behaviors are based on [20]. $Amplitude_{OutputElectricalSignal} = ConversionFactor * PeakPower_{InputPulse}$
Detector, Single Photon (SPD)	The Single Photon Detector is a one port, unidirectional optical component used to detect optical pulses that are weak. Optical pulses pass into the optical receiver port, but are detected only during "gated" periods. Once detected, electrical signals are generated and transmitted through the electrical output port. Note that optical pulses can only pass into the Single Photon Detector in the forward direction, not the reverse direction. For a survey of SPD technologies, please see [21, 22]. $D = \eta_{Detector} * \eta_{channel}$, with η=efficiency
Half-Wave Plate	The Half-Wave Plate is a two port, bidirectional optical component used to create a phase shift that rotates the polarization of the linearly polarized light. The orientation of the output pulse is shown in the equation below, where orientation is α, elipticity is φ, and the device offset angle is γ. The modeled behaviors are based on [26, 27, 28, 29, 30]. $Orientation_{\mathbf{Output}} = arccos\, P(\gamma, \alpha, \varphi)$ $P(\gamma, \alpha, \varphi) = \frac{1}{2}\sqrt{2 + 2\cos(2\alpha)\cos(4\gamma) + 2cos(\varphi) * sin(2\alpha) * sin(4\gamma)}$
In-line Polarizer	The In-line Polarizer is a two port, bidirectional optical component used to polarize optical pulses. Light that has polarization orthogonal to the input pulse polarization of light is blocked and only those of the same polarization are transmitted. The amplitude of the output pulse is calculated using the amplitude of the input pulse, insertion loss, and various angle measurements, as shown in the first equation below. The polarization of the output pulse is also given in the second equation below. Note that orientation is α, ellipticity is φ, and the device offset angle is γ. The modeled behaviors are based on [31, 32, 33]. $Amplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-(InsertionLoss)}{10}}}$ $* \sqrt{(\cos(\alpha) * \cos(\gamma))^2 + (\sin(\alpha) * \sin(\gamma))^2 + (2 * \cos(\alpha) * cos(\gamma) * \sin(\alpha) * sin(\gamma) * \cos(\varphi))^2}$ $Polarization_{Output} = \begin{pmatrix} (cos(\gamma))^2 & \cos(\gamma) * sin(\gamma) \\ \cos(\gamma) * sin(\gamma) & (sin(\gamma))^2 \end{pmatrix}$

Isolator	The Isolator is a two port, bidirectional optical component used to transmit optical pulses in the forward direction and highly attenuate optical pulses passing in the reverse direction, known as isolate pulses. The amplitude of the isolate pulse is calculated using the amplitude of the input pulse, insertion loss, and isolation loss, as shown in the first equation below. The amplitude of the output pulses is calculated just using the amplitude of the input pulse and insertion loss, as shown in the second equation below. The modeled behaviors are based on [34, 35]. $Amplitude_{Isolated} = Amplitude_{Input} * \sqrt{10^{\frac{-(InsertionLoss+IsolationLoss)}{10}}}$ $Amplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-InsertionLoss}{10}}}$
Laser	The Laser is a one port, unidirectional optical component used to generate coherent optical pulses, which are characterized as either a timing pulse or a random pulse. Timing pulses are strong coherent pulses, while random pulses are weak coherent pulses. Both types of pulses are transmitted to the electrical control port. Note that optical pulses cannot pass into the Laser. The modeled behaviors are based on [36, 37].
Optical Switch 1x2	The Optical Switch 1x2 is a three port, unidirectional optical component used to route optical pulses to the desired output port. This routing switches the output pulses in order to monitor the output power levels. When a pulse is transmitted to the desired output port, a reflection of the pulse is generated. The amplitude of the desired output pulse is calculated using the amplitude of the input pulse and insertion loss, as shown in the first equation below. The amplitude of the isolated pulse is calculated using the same variables as the amplitude of the desired output pulse and also the isolation loss, as shown in the second equation below. Note that optical pulses can only pass into the Optical Switch 1x2 in the forward direction, not the reverse direction. The modeled behaviors are based on [38, 39, 40, 41, 42]. $DesiredAmplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-(InsertionLoss)}{10}}}$ $IsolatedAmplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-(IsolationLoss)}{10}}}$
Polarization Controller	The Polarization Controller is a two port, unidirectional optical component used to compensate for the polarization drift in orientation and ellipticity that occurs from the sender to the receiver by adjusting the polarization. The signal orientation of the output pulse is calculated based on whether the compensation lag effect flag is on or off and whether the maximum adjustment angle is less than or greater than/equal to the desired orientation minus the signal orientation of the input pulse. Note that optical pulses can only pass into the Polarization Controller in the forward direction, not the reverse direction. The modeled behaviors are based on [45, 46, 47, 48, 49, 50]. $Corrected\ Polarization \cong Reference\ \ Polarization\ (inputsignal)$
Polarization Maintaining Fiber Channel	The Polarization Maintaining Fiber Channel is a two port, bidirectional optical component used to propagate optical pulses passing into the primary port through the fiber. A small amount of attenuation is occurred, but the polarization state is maintained. The amplitude of the output pulse is calculated using the amplitude of the input pulse, insertion loss, and fiber loss, as shown in the equation below. The modeled behaviors are based on [51, 52, 53, 54]. $Amplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-(InsertionLoss+FiberLoss)}{10}}}$ $FiberLoss = Loss * Length/1000$ $Length = Length + (Length * TEC * (Temp_{Current} - Temp_{Initial}))$
Polarization Modulator	The Polarization Modulator is a two port, bidirectional optical component used to modify the polarization of optical pulses. Qubits are encoded by modifying the orientation to the desired angle. The amplitude of the output pulse is calculated using the amplitude of the input pulse and insertion loss, as shown in the equation below. The modeled behaviors are based on [43, 44]. $Amplitude_{Output} = Amplitude_{Input} * \sqrt{10^{\frac{-(InsertionLoss)}{10}}}$
Wave Division Multiplexer	The Wave Division Multiplexer (WDM) is a passive, three port bidirectional optical component used to combine (or split) different wavelengths of light. When operating as a splitter, the WDM has one input port and two output ports where two co-propagating wavelengths are separated onto individuals fibers. When operating as a combin-

	er, the WDM has two input ports and one output port where two input wavelengths are joined on a single fiber. WDNs have many configurations, including free-space dichroic mirrors (using collimating elements to launch from and focus onto the fiber ends) and in-line fiber devices. In commercial applications, WDMs are frequently used for add/drop filters and signal amplification (i.e., erbium doped fiber amplifier (EDFA) pumping). Although advanced WDM devices exist which can combine many wavelengths, we are most interested in the multiplexing of two wavelengths, namely the primary telecommunication wavelengths 1310 and 1550 nm. The WDM requires single-mode input and output fibers. The modeled behaviors are based on [56, 57, 55].

APPENDIX REFERENCES

[1] Thor Labs, "Fixed Fiber Optic Attenuators, Single Mode," [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1385. [Accessed 13 06 2014].

[2] Newport, "Fixed Fiber Optic Attenuator," [Online]. Available: http://www.newport.com/Fixed-Fiber-Optic-Attenuator/835678/1033/info.aspx#tab_orderinfo. [Accessed 13 06 2014].

[3] OZ Optics, "Fixed attenuators and attenuating fiber patchcord," [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/DTS0030.pdf. [Accessed 13 06 2014].

[4] Pacific Interconnections, "Fiber Build Out Attenuators," [Online]. Available: http://www.pacificinterco.com/attenuators/fiber-optic-attenuator.htm. [Accessed 13 06 2014].

[5] OPLINK, "Electronically Variable Optical Attenuators," 2014. [Online]. Available: http://www.oplink.com/pdf/EVOA-S0012.pdf .

[6] Lightwaves 2020, "Liquid crystal based variable optical attenuation for open-loop architecture," 2014. [Online]. Available: http://www.amstechnologies.com/fileadmin/amsmedia/downloads/1073_VOA-LowTDL.pdf .

[7] OZ Optics, "Electronically controlled variable fiber optic attenuator," 2014. [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/DTS0010.pdf.

[8] Newport, "Tunable bandpass fiber optic filter," 2014. [Online]. Available: http://www.newport.com/Tunable-Bandpass-Fiber-Optic-Filter/835502/1033/info.aspx#tab_orderinfo.

[9] Thor Labs, "NIR bandpass & laser line Filters 700 - 1650 nm center wavelength," 2014. [Online]. Available: http://www.newport.com/Tunable-Bandpass-Fiber-Optic-Filter/835502/1033/info.aspx#tab_orderinfo.

[10] AFW Technologies, "Fiber Optic Band Pass Filter," 2014. [Online]. Available: http://www.afwtechnologies.com.au/band_pass_filter.html.

[11] Gould Fiber Optics, "High isolation fiber optic wavelength division multiplexers," 2014. [Online]. Available: http://www.gouldfo.com/highisolationwdm.aspx#specs.

[12] Brevetti, "Tension-tuned acousto-optic bandpass filter". Patent US 6647159 B1, 2002.

[13] Edmund Optics, "What are Beamsplitters?," 2014. [Online]. Available: http://www.edmundoptics.com/technical-resources-center/optics/what-are-beamsplitters/.

[14] OZ Optics, "Beam splitters/combiners," 2014. [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/DTS0095.pdf.

[15] Edmund Optics, "Cube Beamsplitters," 2014. [Online]. Available: http://www.edmundoptics.com/optics/beamsplitters/cube-beamsplitters/.

[16] Thor Labs, "Optical Beamsplitters," 2014. [Online]. Available: http://www.thorlabs.com/navigation.cfm?guide_id=18 .

[17] Thor Labs, "Fiber-Based Polarization Beam Combiners / Splitters, 1 SM and 2 PM Ports," 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6673.

[18] Thor Labs, "Variable Polarization Beamsplitter Kit," 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=316.

[19] DPM Photonics, "Specifications," 2014. [Online]. Available: http://www.dpmphotonics.com/product_detail.php?id=170 .

[20] Thor Labs, “Calibrated Photodiodes,” 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=2822.

[21] R. H. Hadfield, "Single-photon detectors for optical quantum information applications," Nature photonics , vol. 3, no. 12, pp. 696-705, 2009.

[22] M. D. Eisaman, J. Fan, A. Migdall and S. V. Polyakov, “Invited review article: Single-photon sources and detectors,” Review of Scientific Instruments , vol. 82, no. 7, p. 071101, 2011 .

[23] Gould Fiber Optics, “Fiber Optic Circulators,” 2014. [Online]. Available: http://www.gouldfo.com/circulator.aspx.

[24] Thor Labs, “Single Mode Fiber Optic Circulators,” 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=373 .

[25] Thor Labs, “InGaAs Avalanche Photodetectors,” 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=4047.

[26] Newport, "Zero-order precision wave plates," 2014. [Online]. Available: http://www.nxtbook.com/nxtbooks/newportcorp/resource2011/#/800.

[27] Wolfram, "Polarization of Light through a Wave Plate," 2014. [Online]. Available: http://demonstrations.wolfram.com/PolarizationOfLightThroughAWavePlate/.

[28] OZ Optics, “Polarization rotators/controllers/analyzer,” 2014. [Online]. Available: http://www.icwic.com/icwic/data/pdf/cd/cd069/Polarizers,%20FO/a/111999.pdf.

[29] OZ Optics, “Polarization rotators/controllers/analyzers,” 2014. [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/DTS0072.pdf.

[30] Newport, "Polarization," 2014. [Online]. Available: http://www.newport.com/Polarization/144921/1033/content.

aspx.

[31] Thor Labs, "In-Line Fiber Optic Polarizers," 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=5922.

[32] Newport, "Fiber Optic In-Line Polarizers," 2014. [Online]. Available: https://www.newport.com/Fiber-Optic-In-Line-Polarizers/849607/1033/info.aspx#tab_Specifications.

[33] OZ Optics, "Polarizers - Fiber Optic," 2014. [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/DTS0018.pdf.

[34] Thor Labs, "IR Fiber Optic Isolators with SM Fiber (1290 - 2010 nm)," 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6178.

[35] K. W. Chang and W. V. Sorin, "High-performance single-mode fiber polarization-independent isolators," Optics letters , vol. 15, no. 8, pp. 449-451, 1990.

[36] Thor Labs, "Coherent Sources," 2014. [Online]. Available: http://www.thorlabs.com/navigation.cfm?guide_id=31.

[37] L. O. Mailloux, M. R. Grimaila, D. D. Hodson and C. McLaughlin, "Modeling Continuous Time Optical Pulses in a Quantum Key Distribution Discrete Event Simulation," in International Conference on Security and Management SAM'14, 2014.

[38] oeMarket, "1x1/1x2 fiber optical switch," 2014. [Online]. Available: http://www.oemarket.com/catalog/product_info.php/1x11x2-fiber-optical-switch-p-81?osCsid=4618a22aaa4eae756f73ad2fd2a293b1.

[39] DiCon fiberoptics, inc., "Optical switches," 2014. [Online]. Available: http://www.diconfiberoptics.com/products/prd_switches.php.

[40] DiCon fiberoptics, inc., "MEMS 1xN singlemode optical switches," 2014. [Online]. Available: http://www.diconfiberoptics.com/products/?prod=0044&menu=swt&sub=0.

[41] Thor Labs, "OSW12-1310-SM - Vis/NIR MEMS 1x2 Switch, 1310 nm, SMF," 2014. [Online]. Available: http://www.thorlabs.com/thorProduct.cfm?partNumber=OSW12-1310-SM.

[42] Luminos, "Single 1x2 fiber optic switch," 2014. [Online]. Available: http://luminos.com/products/switches/s12/?gclid=CIWE-4f29LMCFYZM4Aod4F4AWQ.

[43] Thor Labs, "Polarization Optics," 2014. [Online]. Available: https://www.thorlabs.com/navigation.cfm?guide_id=8.

[44] OZ Optics, "Polarizers - fiber optic," 2014. [Online]. Available: http://www.google.com/url?sa=t&rct=j&q=&esrc=s&frm=1&source=web&cd=1&cad=rja&uact=8&sqi=2&ved=0CB0QFjAA&url=http%3A%2F%2Fwww.ozoptics.com%2FALLNEW_PDF%2FDTS0018.pdf&ei=7eSeU7uZBo7esATb9YDgBQ&usg=AFQjCNGAvNRioVcR9E3I6S-qHupXH3uWCQ.

[45] Thor Labs, "Deterministic Polarization Controller - DPC5500," 2014. [Online]. Available: http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=930.

[46] Phoenix Photonics, "Fiber polarization controller," 2014. [Online]. Available: http://www.phoenixphotonics.com/documents/polarizationcontroller_01202.pdf.

[47] Phoenix Photonics, "Electronic In-Line Fiber Polarization Controller - EPC With Optional Microprocessor Based PC Interface," 2014. [Online]. Available: http://www.phoenixphotonics.com/website/products/Electronically_Controlled_Polarization_Controller.htm.

[48] OZ Optics, "Electrically driven polarization controller-scrambler," 2014. [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/DTS0011.pdf.

[49] Fiber Logix, "Electronically addressed Polarization controller," 2014. [Online]. Available: http://www.fiberlogix.com/Passive/Electronically%20addresses%20Polarization%20controller.html.

[50] General Photonics, "Dynamic Polarization Controller/Scrambler," 2014. [Online]. Available: http://www.ainnotech.com/pdf/GP-Modules-Polarization%20Management-Dynamic%20Polarization%20Scrambler%20Controller.pdf.

[51] Corning, "PANDA PM Specialty Optical Fibers," 2014. [Online]. Available: http://www.corning.com/WorkArea/showcontent.aspx?id=18341 .

[52] OZ Optics, "http://www.ozoptics.com/ALLNEW_PDF/ART0001.pdf," 2014. [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/ART0001.pdf.

[53] R. P. Encyclopedia, "Polarization-maintaining Fibers," 2014. [Online]. Available: http://www.rp-photonics.com/polarization_maintaining_fibers.html.

[54] Thor Labs, "Polarization-Maintaining FC/PC Fiber Optic Patch Cables," 2014. [Online]. Available: http://www.thorlabs.com/search/thorsearch.cfm?search=polarization+maintaining+fiber+optics.

[55] Pacific Interconnections Group, “Fiber Optic WDM,” 2014. [Online]. Available: http://www.pacificinterco.com/Splitters_N_Couplers/1310-1550-WDM.htm.

[56] OZ Optics, “Wavelength Division Multiplexers,” 2014. [Online]. Available: http://www.ozoptics.com/ALLNEW_PDF/DTS0089.pdf.

[57] Gould Fiber Optics, “Fiber Optic Wavelength Division Multiplexers (WDMs),” 2014. [Online]. Available: http://www.gouldfo.com/wdm.aspx.

7. Using the Discrete Event System Specification to Model Quantum Key Distribution System Components

Morris, J.D., Grimaila, M.R., Hodson, D.D., McLaughlin, C.V. & Jacques, D.R. Using the Discrete Event System Specification to Model Quantum Key Distribution System Components.

Published in: **Under review at The Journal of Defense Modeling and Simulation, Jul. 2014**

17 Pages

Using the Discrete Event System Specification to Model Quantum Key Distribution System Components

Jeffrey D. Morris, Michael R. Grimaila, *Senior Member*, *IEEE*, Douglas D. Hodson, Colin V. McLaughlin, and David R. Jacques

Abstract*—In this paper, we present modeling Quantum Key Distribution (QKD) system components using the Discrete Event System Specification (DEVS) formalism. The DEVS formalism assures the developed component models are composable and exhibit temporal behavior independent of the simulation environment. These attributes enable users to assemble and simulate any collection of compatible components to represent complete QKD system architectures. To illustrate the approach, we introduce a prototypical "prepare and measure" QKD system, decompose one of its subsystems, and present the detailed modeling of the subsystem using the DEVS formalism. The developed models are provably composable and exhibit behavior suitable for the intended analytic purpose, thus improving the validity of the simulation. Finally, we discuss issues identified during the verification of the conceptual DEVS model and discuss the impact of these findings on implementing a hybrid QKD simulation framework.

***Index Terms*—Conceptual Modeling; Discrete Event Simulation; Discrete Event System Specification; Modeling and Simulation, Quantum Key Distribution**

I. Introduction

CRYPTOGRPAHY, the practice and study of techniques for securing communications between two authorized parties in the presence of one or more unauthorized parties, is the centerpiece of a centuries old battle between code maker and code breaker [1]. Historically, only financial, government, and military applications used cryptography; but today much of modern society depends on cryptography to provide security services including confidentiality, integrity, authentication, and non-repudiation [2]. While there are many types of cryptography, only the One-Time-Pad (OTP) symmetric key algorithm is "information-theoretically secure" [3], [4]. All other forms of cryptography are breakable if the adversary has enough cipher text, computational resources, and time [5]. Despite its strength, the OTP is not in common use because of the large amount of secret key material required for its proper use. These keys require random generation, length equal to the message, and single use. These requirements impose significant limitations on use of the OTP in most applications due the costs involved with secure key generation and distribution.

Quantum Key Distribution (QKD) is a technology that offers the means for two geographically separated parties to create a shared secret key [6]. QKD is unique in its ability to detect any third-party eavesdropping on the key exchange, assuring the secrecy of the key. This is possible due to the fundamental laws of quantum mechanics which ensures any third-party eavesdropping on the quantum channel introduces detectable errors. Combining a QKD-generated key with the classical OTP realizes an "unconditionally secure" cryptosystem.

A. The Need for QKD Simulation

However, QKD is a developing technology and has not been thoroughly studied from a systems-level perspective. QKD systems contain non-ideal components that differ, sometimes significantly, from the ideal components specified during the original conceptual system design. Therefore, there is a need to develop an efficient integrated modeling and simulation capability to understand the impact non-ideal components have on the performance and security of different QKD system architectures.

There exist few QKD simulations beyond those that model specific hardware or situations. An example is the Austrian Institute of Technology's AIT QKD Software project [7] that attempts to model an entire QKD network but is based mainly on their entanglement QKD hardware. An extensive literature search over several years revealed no other system-level QKD modeling & simulation (M&S) efforts.

To address this shortcoming, we have developed a modular simulation framework, named *qkdX*, which provides users the capability to model rapidly, simulate, and study QKD system architectures. This simulation capability provides hybrid functionality as it abstracts continuous-time QKD system

Paper submitted May 21, 2014. This work was supported by the Laboratory for Telecommunication Sciences [grant number 5743400-304-6448].

Jeffrey D. Morris is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: jeffrey.morris@afit.edu).

Michael R. Grimaila is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: michael.grimaila@afit.edu).

Douglas D. Hodson is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: douglas.hodson@afit.edu).

David R. Jacques is with the Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-7765 USA (email: david.jacques@afit.edu).

Colin McLaughlin is a Research Physicist at the Naval Research Laboratory, Washington, D.C. 20375 USA (email: colin.mclaughlin@nrl.gov).

signals (e.g., electrical signals and optical pulses) into a representation suitable as events in a Discrete Event Simulation (DES) environment [8]. A continuous-time simulation of a complete QKD system is infeasible due to the enormous number of optical pulses generated during system operation and necessitates the use of DES.

The researchers, in consultation with Subject Matter Experts (SMEs) in the optical physics and electrical engineering domains, determined the abstraction necessary for each signal model. The abstraction enables a system-level simulation where signals propagate through the system as discrete events, but can be reconstructed into a continuous-time representation when mathematical operations or transformations of the signals are required. The details of the optical pulse model and related mathematical transforms are outside the scope of this paper. Instead, we focus on the proper modeling of the temporal behavior and internal "state" of QKD system components.

To capture this temporal behavior and the state of components, we use the Discrete Event System Specification (DEVS). In the past, DEVS has been used to model high-level architectures, hybrid-systems, cell-spaces, distributed supply chains, test & evaluation, forest fires, environmental systems, building performance models, and other problem spaces [9-16]. This paper presents, to our knowledge, the first use of DEVS to model optical components.

B. The Need for Validity in Simulation

Model validity is a necessary condition for the credibility of simulation results [17]. Model validation, according to Balci, is "substantiating that the simulation model, within its domain of applicability, behaves with satisfactory accuracy consistent with the study objectives" [18]. Model validation is the comparison of model behavior to the behavior of the system under study when both are responding to identical input conditions [19]. Model testing identifies failures, corrects them, and then retests to the required accuracy and behavior [17], [18].

Complete testing of a model throughout its solution space is not possible. Such testing is cost and time prohibitive; instead testing continues until attaining sufficient confidence in the model for its intended purpose [19]. Fig. 1 shows the relationship between value, cost and model confidence. As user confidence in the model increases there is a corresponding logarithmic increase in the value of the model, but the cost increases exponentially. Eventually the gain in value is negligible but costs continue to increase steeply.

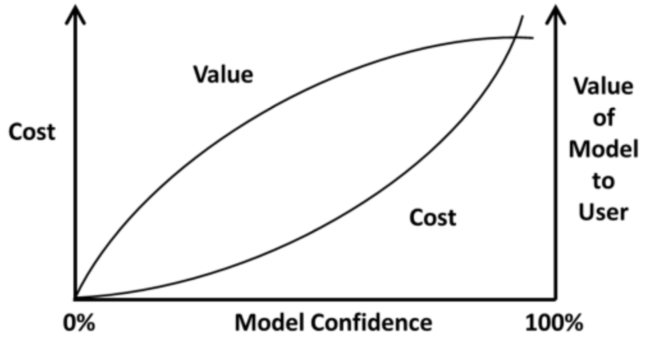


Fig 1. Model Confidence [19].

Validity and model confidence relate closely. The better the belief the model accurately represents the system under study, the higher the validity of the model. Higher validity suggests a higher confidence in the model being useful for its purpose. Exhaustive testing brings higher confidence, but at much greater cost. Our research is focused upon exploring a means to increase model validity without the need to exhaustively test the entire solution space.

Specifically, this research focuses on how using the Discrete Event System Specification (DEVS) formalism and concept model validity theory 0increases the validity of a QKD system-level simulation. To achieve this goal, we present modeling QKD system components using the DEVS formalism [20]. Representing component behavior using the DEVS formalism ensures the developed conceptual models exhibit composibility and deterministic temporal behavior independent of the simulation environment. Additionally, we identified unforeseen benefits that arise when using a strict modeling formalism.

The remainder of this paper is organized as follows. Section II reviews the DEVS formalism and conceptual modeling. Section III explains the use of DEVS in our modeling effort. We introduce a prototypical QKD system architecture and decompose its "classical pulse generator" (CPG) subsystem in Section IV. Section V presents modeling the CPG subsystem using DEVS atomic and coupled models. Section VI presents our findings, reviews issues identified during simulation and verification of the conceptual model, and discusses the impact of these findings on implementing the hybrid simulation framework. Finally, we provide concluding remarks in Section VII.

II. Discrete Event System Specification and Conceptual Model Validity

A. What is DEVS?

Zeigler first proposed the DEVS in 1976 as a hierarchal formalism for decomposing complex discrete-event systems into simple components, leading to *well-defined* behavior of the overall model [21]. Zeigler states "…the set of all dynamic systems is taken as a well-defined class in which each system has a set of input time segments, states, state transitions and output time segments." DEVS interprets the dynamic systems as sets and functions and sets conditions needed for a well-defined specification [22]. DEVS defines system behavior, syntax, and structure, enabling modularity within a DES by building complex systems from simple (*atomic*) components.

It uses dynamical systems theory as a means to canonically represent system behavior and provide provable *closure under coupling* (also known as *composability*) [23], [24]. DEVS provides an efficient way to represent complex systems in a hierarchal manner to create *coupled* (compound) modules, or subsystems, to create complete system models. The theory states "…the dynamic system specified by a coupled model can be represented as (more technically, is behaviorally equivalent to) an atomic DEVS system" [22].

DEVS separates the model from the source system and the simulator. This allows for a conceptual model not tied to any particular simulator and creates a bounded source with finite inputs and outputs. Hoffman describes DEVS as a "theoretical confirmation" of transformations between different techniques & tools for modeling systems [25].

Fig. 2 shows a representation of basic entities in modeling and simulation. The source system (i.e., the system under study) couples with a database of behaviors derived from a set of inputs. This *experimental frame* defines the system of interest the modeler is trying to capture [26] and allows the modeler to create a conceptual model for the system under study. Zeigler links the experimental frame and the model with a *modeling relation* and the model and the simulation with a corresponding *simulation relation*. The first relation describes how well observed system behavior matches model-generated behavior (validity) and the second with how well the simulation executes model instructions (verification). Note the model is the bridge between the system and the simulator.

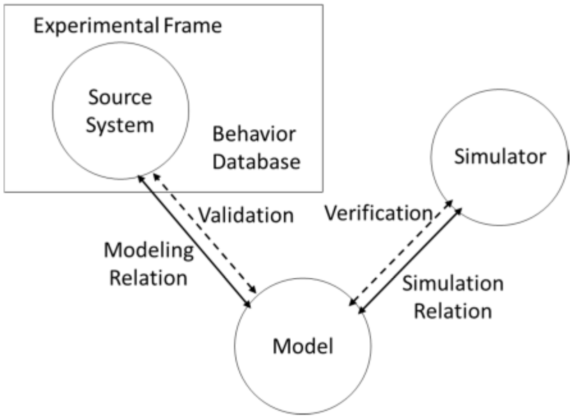


Fig 2. Basic entities in modeling and simulation, adapted from [26].

DEVS provides a concise way of describing the inputs, states, outputs, and timing of a system under study. It is a formal language used to define the conceptual model of the system [27].

A DEVS atomic model has sets of inputs, states, and outputs along with transition and output functions to construct a representation of any dynamic system [22]. Fig. 3 provides a graphic representation of DEVS modeling state transitions in response to incoming events. We start in an initial state *s* and remain in that state for some time advance period *ta*. Once *ta* is reached, the system may output some value *y* per the output function $\lambda(s)$. Immediately after the output, the system goes through the internal transition, δ_{int} (s), based on the current state. The system changes to state, s', which becomes the new state *s* and the cycle starts over. If the state receives an external disturbance during the time *ta*, at some elapsed time since the last transition, *e*, the system undergoes an external transition δ_{ext} (s,e,x). The external transition uses the existing state, *s*, the time elapsed *e* and the input values *x* to determine the new state, *s*, and the cycle starts anew.

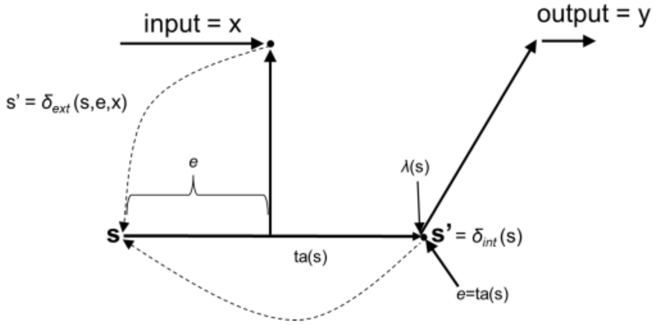


Fig 3. DEVS sequence diagram, derived from [28].

While there are different types of DEVS, Parallel-DEVS has several characteristics necessary to model QKD components. The Parallel-DEVS formalism specifies the following about each atomic model [29]:

- Ports between models are represented explicitly – there can be any number of input and output ports.
- Atomic DEVS models can handle *bags* of inputs and outputs.
- A bag can contain many elements with possibly multiple occurrences of its elements.
- The external transition function handles inputs of bags.
- The output function can generate a bag of outputs.
- The confluent transition function, $\delta_{con}(s, ta(s), x)$ decides the processing order of simultaneous external and internal events.

In this paper, we make use of Parallel-DEVS as it provides the unique abilities of queues, necessary to handle multiple arriving optical packets, and the confluence transition function, necessary for handling simultaneous events [30].

B. *Why Use DEVS?*

The DEVS formalism ensures well-defined component temporal behavior and provable closure under coupling (i.e., composability), allowing for easier verification of correctness of component compositions, and improving the validity of system representations. Thus, for our purpose, we are ensured the temporal behavior of high-level QKD system representations reflects the dynamics of its constituent parts. In other words, the burden of verifying high-level system dynamics focuses on correct modeling of constituent parts (components); once accomplished, we can assemble high-level representations for study with confidence the assembly process itself has not introduced unforeseen effects or anomalies.

1) *Conceptual Modeling & Validity*

This research focuses on the use of DEVS conceptual modeling to capture the behavior of components found in the optical path of a QKD system. The question remains as why do this? Conceptual modeling is the process of determining what to model to be useful [31]. This process has been described as conceptualization of real-world referents that varies from modeler to modeler [25] whereas Robinson describes the conceptual model as “non-software specific description of the simulation model that is to be developed” [32]. Deciding the appropriate “wrongness” (abstraction), agreement on the model, and model validation are some of the objectives of conceptual modeling [31]. Though conceptual modeling has been described by some as more art than science [33], Robinson provides a framework for conceptual modeling and lists five activities in a process for conceptual modeling [32]:

- Understanding the problem situation.
- Determining the modeling and general project objectives.
- Identifying the model outputs (responses).
- Identify the model inputs (experimental factors).
- Determining the model content (scope and level of detail), identifying any assumptions and simplifications.

This is where the usefulness of the DEVS formalism becomes apparent. It provides a mathematically-proven process to work through the objectives. Using DEVS to model the components for the QKD demonstration architecture gives a way to meet the objectives of conceptual modeling while accomplishing the steps in the modeling process. DEVS forces the modeler to have a deep understanding of behavior and timing which in turn requires understanding the problem, the modeling objectives, and capturing inputs and outputs.

How hard it is to distinguish between the model and the source system is the question of *validity* [34]. The harder it is to distinguish between the model and the source system in the experimental frame, the greater the validity. Note that a model’s validity only applies to the experimental frame of interest. Change the frame and you change the validity of the corresponding model.

As mentioned, it is cost-prohibitive to check every possible model combination and trying to validate the entire model space by model checking or theorem-proving approaches is nearly impossible once a model has many connections or interactions [35]. DEVS allows for increased model validity without having to check the entire model space.

2) *How Does DEVS Increase Validity?*

Since validity is a measure of “closeness,” how does DEVS increase validity? Using DEVS forces the modeler to have a deep understanding of the modeled behavior because the formalism requires it. This lessens or eliminates undesired, unexpected or emergent behavior. By knowing exactly how the model behaves, it can be matched and changed to the observed behavior of the source system.

DEVS provides three levels of validity for conceptual models: replicative, predictive and structural [34]. Each level of validity meets the requirements of the previous level(s). The model and system achieve replicative validity if their behaviors agree to acceptable levels for all experiments captured in the behavioral database for the experimental frame. The second, predictive, requires the model generate the same output as the system for any experiment not captured in the experimental frame database. This requires the model to be able to be set into the same state as the system for the experiment, for any acceptable starting state. Finally, structural validity requires the model and system have a corresponding step-by-step, component-by-component transition through all possible states. Any model properly using DEVS achieves all three of these states, making it harder to distinguish between the model and system under study, and increasing its validity, per the earlier definition.

Another consideration is the temporal behavior of the source system. In many discrete-event simulators, the underlying implementation of the simulator influences the behavior of the model when multiple simultaneous events occur. DEVS addresses this problem by providing a confluence function to express the behavior of the model for these situations. This means a DEVS model exhibits the same behavior on any DEVS-compliant simulator and if the simulation relation is sound, the behavior can be replicated on any DES.

Lastly, DEVS promotes sound model development when used as an intermediate step towards developing a large DES model using a non-DEVS-compliant simulator. For example, some discrete-event simulators schedule events in the future for convenience [36]. This situation can cause problems; DEVS avoids this as there are no future events in the formalism. There are only two types of time in DEVS: the time advance (*ta*) and the elapsed time (*e*). This forces the modeler to carefully consider time and input interaction on all states.

III. Using DEVS to Model QKD System Components

A. *Methodology*

1) *The Modeling Process*

The QKD modeling process began with discussions between the research team and the SMEs in the areas of optical physics, quantum physics, electrical and software engineering. These discussions led to an agreement on simulation objectives, what needs modeling, the fidelity necessary, assumptions, and simplifications needed or wanted within the model. The results were a small-scale proof-of-concept simulation using the OMNeT++ DES [36] to show the basic premises were sound and selecting DEVS to build the conceptual model [8].

The optical physics SME created a mathematical model for each of the optical components that captured parameters and behavior believed necessary for the model. Model creation used a combination of data measured during laboratory experiments in conjunction with component data sheets and existing reference literature. Creating and verifying the correctness of the developed math models used Mathematica, a well-known math computation software package [37].

The next step was to transform the mathematical model into a DEVS pseudocode model. The modeler reviewed the math models to understand the necessary transformation functions, reviewed quantum and optical physics literature and consulted with the SMEs to understand the required component behavior. Product literature for existing physical components provided additional information for acceptable component input and output ranges. The DEVS models captured this information using phases, states and transitions and submitted the component models back to the optical SME for review.

Once complete, the DEVS pseudocode became the basis for creating the model in a DEVS-compliant simulator, MS4ME [38]. MS4ME is a product of RTSync (www.rtsync.com), a spin-off from the Arizona Center of Integrative Modeling and Simulation (ACIMS) [39]. MS4ME provides a structured user interface for modeling in the ACIMS DEVS-JAVA language. For each component, the output from the MS4ME simulator was compared against the expected behavior of the DEVS model. This modeling was a check on the DEVS pseudocode and ensured the models met the requirements of the formalism and captured the appropriate behavior. Once checked, the DEVS pseudocode became the basis for the simulation modelers to create the qkdX framework. As shown in Fig. 4, DEVS is the intermediate step between the SME mathematical model and the QKD simulation.

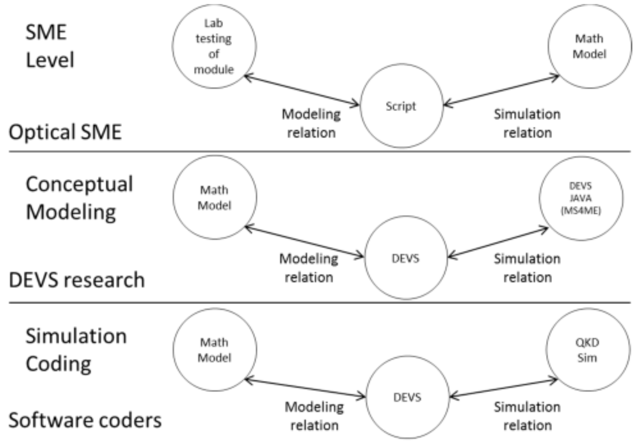


Fig 4. Levels of modeling and simulation.

2) *SME to Conceptual Model to Simulation Cycle*

During the conceptual modeling process, the modeler worked with the software and electrical engineers to capture the hardware and software behavior of QKD devices. A constant review process looked for differences between the proof-of-concept demonstrator and the detailed DEVS models. Once complete, the DEVS models went to the simulation modelers (the software and electrical engineers) for use in adapting the existing proof-of-concept simulation code to agree with the conceptual model. This process is an example of the simulation relation.

The research team held weekly teleconferences and had several site visits in a continuing effort to better understand and model the QKD system. Development of the DEVS modules and translation into the both the MS4ME DEVS simulator and the qkdX simulation framework resulted in the identification of multiple inconsistencies between the representations. These inconsistencies were reconciled to yield canonical behavior between all simulation models.

Throughout this process, the modeler continued to consult with optical SMEs. Each completed model underwent review by optical SMEs to ensure the DEVS model captured the proper behavior and essential parameters. This is an example of the idea of the model relation.

Fig. 5 shows an overview of our research modeling process. The SME generated the mathematical models used to create the conceptual models. Constant two-way communication involved the SME and the simulation modelers in the conceptual modeling process. Once acceptable to the SME, the conceptual models were given to the simulation modelers for translation into the simulation framework. Once again, there was constant communication between the conceptual modeler, the SME and the framework modelers. This circular modeling process ensured acceptance between all parties.

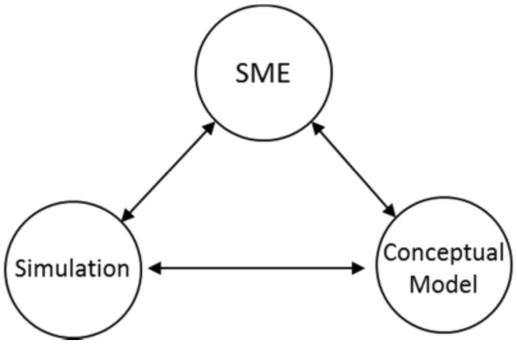


Fig 5. Research modeling process.

IV. A Prototypical Polarization-Based BB84 Prepare and Measure QKD System

Consider the model of a prototypical QKD system that uses a polarization-based, Bennett and Brassard "BB84" prepare and measure protocol shown in Fig. 6 [40]. The QKD system comprises an "Alice" subsystem, a "Bob" subsystem, an authenticated public communications channel, and a quantum communication channel. Due to the complexity of a QKD system, we focus our discussion on decomposition of the Alice subsystem, the Alice quantum module CPG subsystem to illustrate modeling QKD system components using DEVS.

The Alice subsystem responsibilities include producing and encoding photons with candidate secret key bits and sending the photons to the Bob subsystem via the quantum channel. The Bob subsystem receives the encoded photons and decodes them to recover the candidate key bits. Alice and Bob coordinate their system operations by communicating over the authenticated public channel.

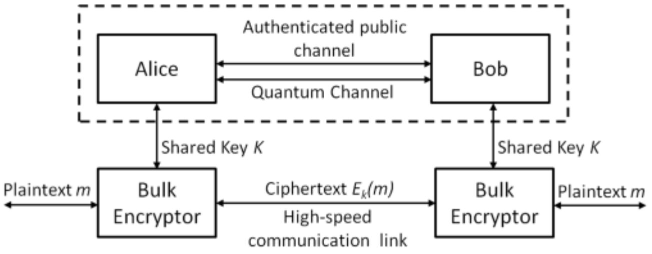


Fig 6. QKD context diagram showing the QKD system and the bulk encryptors using the generated key.

A. Alice Subsystem Decomposition

The Alice subsystem contains several subsystems including a system controller module, a public channel module, a dedicated QKD module, a quantum module, a clock, and a True Random Number Generator (TRNG) as shown in Fig. 7.

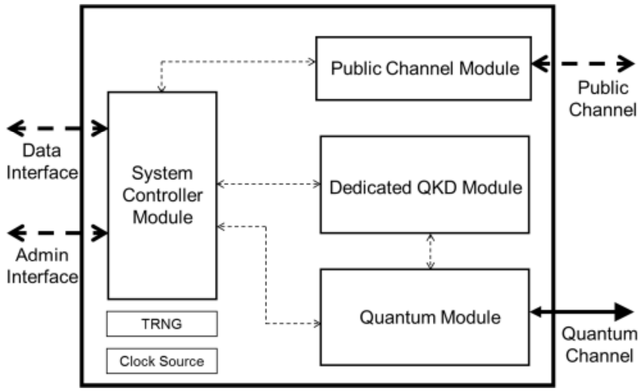


Fig 7. Alice subsystem decomposition.

The Alice system controller module is responsible for controlling the Alice subsystem and serves as the master controller to coordinate operations between Alice and Bob. The public channel module interfaces with the system controller module and provides connectivity to the remote system via the public channel. The dedicated QKD module controls QKD-specific processing such as error detection and correction, sifting, and privacy amplification. The quantum module is responsible for generating the quantum state in optical pulses before sending them to Bob via the quantum channel. The clock source provides reference timing for all synchronous devices. Since the security of a QKD system is a strong function of the randomness of the numbers it generates, a TRNG such as the idQuantique Quantis optical random number generator is typically used to provide the required source of entropy [41].

B. Alice Quantum Module Subsystem Decomposition

The quantum module decomposes into nine different subsystems, Table I shows a brief description of each subsystem function and Fig. 8 illustrates the decomposition.

TABLE I

DESCRIPTION OF ALICE QUANTUM MODULE SUBSYSTEMS

Subsystem	Function
Classical Pulse Generator (CPG)	Generates a multi-photon pulse
Polarization Modulator	Polarizes the photon pulse into the desired polarization
Electronically Variable Optical Attenuator (EVOA)	Creates decoy states to mitigate photon splitting attacks
Fixed Attenuator	Converts classical laser pulses to quantum levels by attenuating to weak-coherent levels
Optical Security Layer	Detects optical probing attacks
Wave Division Multiplexer (WDM)	Multiplexes signal and timing pulses
Timing Pulse Generator	Generates a timing pulse used for synchronization
Switch	Allows generated pulses to be directed to the Output Power Monitor for loop-back testing
Output Power Monitor	Monitors the output optical power

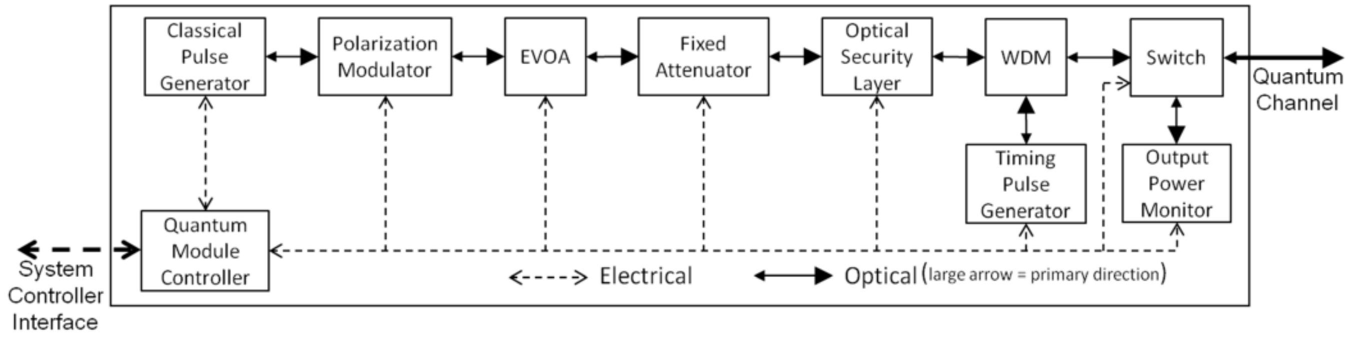


Fig 8. Alice quantum module subsystem decomposition showing internal subsystems and components.

C. Alice Classical Pulse Generator Subsystem Decomposition

The ideal conceptual model of a QKD system specifies polarization-encoded single photons with the desired bit and basis. In reality, reliable on-demand single photon pulse generators are an unrealized technology. Real-world QKD system implementations instead generate a laser pulse containing millions of photons and strongly attenuate the pulse down to statistical sub-photon (quantum) levels. Within the Alice quantum module, the CPG subsystem generates the laser pulses and shifts them into a known polarization. The CPG subsystem contains the components shown in Fig. 9.

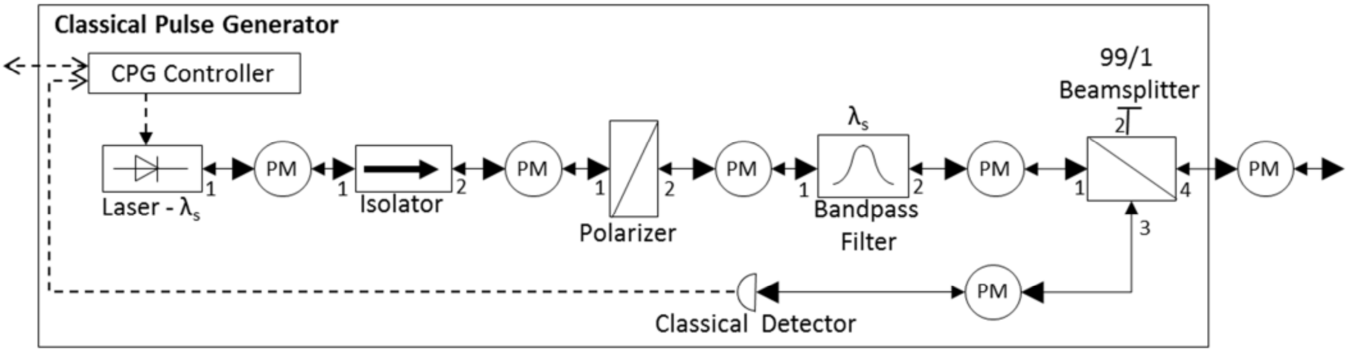


Fig 9. Classical pulse generator subsystem showing internal components.

The CPG subsystem contains a controller, a laser, an isolator, an optical polarizer, an optical bandpass filter, a beamsplitter, a classical detector, electrical interfaces, and interconnecting polarization-maintaining (PM) optical fiber. We first discuss the behavior of the subsystem as a whole and then briefly discuss the behavior of each of the components contained within the CPG.

D. Individual CPG Component Behavior

1) CPG Controller

The controller is an electrical device containing digital and analog circuits responsible for controlling the laser and monitoring the classical detector. It has a bidirectional electrical interface to the quantum module controller, an electrical output to the laser, and an electrical input from the classical detector. It receives commands from the quantum model controller, sends fire commands to the laser, and monitors the health of the laser.

2) Laser

The laser is an electro-optical device which contains an optical oscillator and emits coherent light. It has an electrical input to receive control messages and an optical output to emit generated pulses. Within the simulation, the laser creates optical pulses when it receives a "fire" command from the controller. The laser generates short-duration laser pulses (e.g., 1mW peak intensity with a 500ps duration) containing millions of photons. The output of the laser couples to the input of the isolator via PM fiber.

3) Isolator

The isolator is an optical device with two bidirectional optical ports that passes light in the forward direction while significantly attenuating light moving in the opposite direction. Optical signals arriving at one port propagate to the other port after a defined propagation delay with the attenuation based on the propagation direction. The isolator assures that virtually no light (e.g., reflections or light from external sources) enters the laser. The output of the isolator is coupled to the input of the polarizer via PM fiber

4) Polarizer

The polarizer is an optical device with two bidirectional optical ports allowing light of one polarization to pass while highly attenuating light orthogonal to the passed light. Optical signals arriving at one port propagate to the other port after a defined propagation delay and polarized depending on the polarizer orientation with respect to the connected fiber. The output of the polarizer is coupled to the input of the optical bandpass filter via PM fiber.

5) Bandpass Filter

The bandpass filter is an optical device with two bidirectional optical ports that passes the optical energy in a narrow band around the signal wavelength, λ_S, but strongly attenuates other wavelengths. This ensures that only the appropriate signal wavelength leaves the subsystem while preventing other sources of light from entering the laser. Optical signals arriving at one port propagate to the other port after a defined propagation delay and are attenuated based on the wavelength of the signal. The bandpass filter output couples to port 1 of the beamsplitter.

6) Beamsplitter

The beamsplitter is an optical device used to split a single beam of light into two components. It can also be used to combine two beams of light into one stream. Unlike most of the optical devices, it has four bidirectional optical ports. In the splitting configuration, optical signals arriving at one port are split into two beams, propagating to the appropriate output ports after a defined propagation delay. Common splitting ratios are 50:50, 90:10, and 99:1, but devices exist in almost any ratio. Beams can also be split according to optical wavelength or polarization.

The beamsplitter passes 99% of the pulse through to port 4, leaving the CPG and connecting to the next quantum module subsystem as shown in Fig. 9. Meanwhile, port 3 passes 1% of the pulse on to the classical detector via PM fiber.

7) Classical Detector

The classical detector is an opto-electrical device containing an optical photodiode and support electronics to generate an electrical signal proportional to the power contained in the optical pulse. This signal connects to the controller which stores this information and checks to see if it falls below a predefined threshold. If so, the controller notifies the quantum module controller of an error condition.

8) Polarization-Maintaining Optical Fiber

PM fiber is an optical component used to interconnect optical devices. It has two bidirectional optical ports. Optical signals arriving at one port propagate to the other port after a defined propagation delay. Attenuation is a function of the type and the length of the fiber. PM fiber maintains the polarization of optical signals injected along its fast and slow axes.

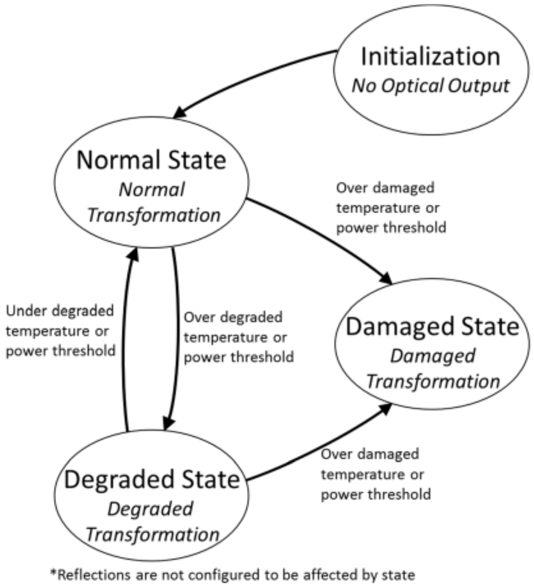
V. Modeling the Alice Classical Pulse Generator Subsystem using the DEVS Formalism

In this section, we discuss features common to all components in the quantum optical path, DEVS modeling of the isolator and the CPG. We selected the CPG as it is unique in containing many types of components found in QKD system simulation: electrical, optical, electro-optical, and opto-electrical.

A. Common Behaviors of Optical Components

All the components that interact with optical signals share some common traits. These include component state, losses to optical intensity, deleting weak packets, environmental ports, and handling multiple pulses. The following section explains these commonalities.

Each modeled optical component is in one of three states: normal, degraded, or damaged as shown in Fig. 10. In the normal state, the component uses a mathematical transform to generate the resulting normal output pulse(s) of the component under normal conditions. When in the degraded state, the component temporarily uses a different transform to generate the resulting degraded output pulse(s), but returns to normal after a predefined time period or condition (e.g. when temperature returns to a normal level). When in the damaged state, the component permanently uses a transform to generate the resulting damaged output pulse(s), if any.


Initialization
No Optical Output
Normal State
Normal Transformation
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
Damaged Transformation
Degraded State
Degraded Transformation
Over damaged temperature or power threshold

*Reflections are not configured to be affected by state
*Electrical signals are not configured to be affected by state

Fig 10. Optical component state transition diagram showing initialization and three states.

Components change states based on the entering optical packet power and from the ambient temperature of the component. If the power in an optical pulse exceeds the defined degraded optical power threshold, it will temporarily enter the degraded state. Similarly, if the power in the optical pulse exceeds the defined damage optical power threshold, it will permanently enter the damaged state. The ambient temperature can also result in a component entering a degraded or damaged state.

When an optical signal arrives at an optical component, a small portion of the light reflects opposite to the light propagation direction. The *return loss* parameter of the component determines the reflected amount, depending on different experimental frames. Certain simulation studies may require the capability to accurately represent reflections, while not desired in other studies. Therefore, we added the capability to turn reflections on and off for each component, reducing the number of events when not needing this fidelity. The pulse not only loses intensity to reflections, a small amount is lost when entering the component. This *insertion loss* parameter may also be turned off and on.

As the optical pulses suffer losses propagating through the system, they are deleted when the optical power drops below a defined minimum threshold. This reduces the number of events and prevents an infinite number of reflections bouncing between two reflecting components. Since fiber couples the optical components together, we chose to implement this function in the fiber.

When dealing with a series of single pulses, as is the case in the normal operation of a QKD system, an optical component typically will have a single pulse propagating through it. However, a robust simulation framework must allow components to be able to handle multiple optical pulses simultaneously arriving and/or propagating through it. Therefore, optical components need a queue to store the multiple pulses. The queue contains port-value pairs and any metadata required to process the pulse. At a minimum, the metadata consists of the arrival port and the time remaining before the transformed optical pulse propagates out of the component. As time transpires in the model, a timer ensures the next transformed pulse processes at the appropriate time and, based on the current component state, the appropriate transform generates the output pulse.

Finally, each component has a separate environmental port so the simulation controller can send environmental messages to mimic temperature variations or physical perturbations (e.g., vibration) in the system. The temperature state has no effect on reflections: a component will reflect optical packets regardless of its current state.

B. Basic Design of a DEVS Model for the Isolator

In this section, we discuss how to model the isolator component using the DEVS formalism. The isolator is representative of most simple optical components and shares their basic behaviors. The isolator allows light to pass in the forward direction while significantly attenuating light moving in the opposite direction. External, internal, confluence, output and time advance functions represent the device, as with all DEVS models [42]. These functions contain logic governing how the component responds to inputs and what and when it will output. DEVS uses a phase within a state as a “marker” to keep track of internal functions within the state. Fig. 11 shows the DEVS phase transition diagram for the isolator and shows the phases (rectangles), the transition arcs (arrows) and any notes for the component.

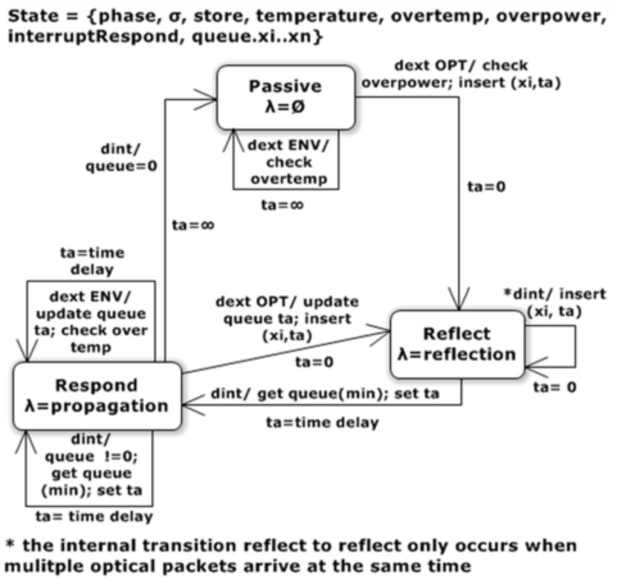

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.xi..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta=∞
ta=0
ta=time delay
dext ENV/ update queue ta; check over temp
dext OPT/ update queue ta; insert (xi,ta)
ta=0
*dint/ insert (xi, ta)
Reflect
λ=reflection
ta= 0
Respond
λ=propagation
dint/ get queue(min); set ta
ta=time delay
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Fig 11. Isolator DEVS phase transition diagram.

The isolator phase transition diagram shows the full state description at the top, which includes the current phase (“Passive”, “Reflect”, “Respond”), the current time advance (σ), variables with names *store*, *temperature*, *overtemp*, *overpower*, *interruptRespond* and the port-value pairs in the *queue*. Taken together, these variables allow construction of the full state of the isolator.

The phase transition diagram shows a Passive phase, where the component awaits input. This phase has a time advance (*ta*) of infinity, meaning it will never reach the point of having an output or internal transition. Once an external event occurs, either an optical packet or an environmental message arriving, the device responds through an external transition. A self-loop labeled with “dext ENV/check overtemp” (external environmental) represents received environmental messages. The device stores the new temperature and checks against the damaged or degraded temperature thresholds, setting the *overtemp* state variable to true if reaching either threshold. The time advance for the follow-on phase (e.g., returning back to Passive) is on the other side of the arc; in this case showing “ta=∞” as the time advance for the Passive phase is infinity.

If an optical packet arrives, different external transition logic executes, this time checking to see if the arriving optical power exceeds the degraded or damaged thresholds. Since DEVS collects all messages arriving at the same time into a single message bag as an unordered collection, the external transition from the Passive phase iterates through the bag, checking for the overpower conditions and placing the events in the queue, finally selecting one at random for reflection. Each optical packet enters the queue as event *xi* with some time *ta*, the time advance the packet will remain in the component. The only choices for Passive phase external events are an environmental or optical message; the isolator ignores any other event that arrives.

The transition arc between the Passive and Reflect phases with “dext OPT/check overpower; insert (xi,ta)” represents the Reflect phase external optical event and shows the *ta* for the Reflect phase is equal to zero. This phase has two external optical events, once coming from the Passive phase and one from the Respond phase.

In DEVS, the output function λ executes only before an internal transition. The Passive phase does not have an output function, as the *ta* for the phase is infinity, meaning there will never be an internal transition. The “λ =0” in the Passive phase rectangle shows this. The Reflect phase output is the optical packet reflection and the Respond phase output is the propagated optical packet.

The Reflect phase has two possible choices for the internal transition, *dint*, with the note at the bottom specifying the self-loop “dint/insert(xi,ta)” executes only when multiple optical packets arrive at the same time. If more packets need reflecting, the self-loop internal transition is called. The Reflect phase checks the queue for any packet not yet reflected, selects it and emits it. As noted before, the time advance of this phase is zero, so this takes place in instant simulation time until complete. This follows the optical SME guidance that reflections must take place before any other operation.

Once done with all reflections, the Reflect phase enters the other internal transition, which removes the queued packet with the shortest time left in its time delay and sends it to the Respond phase. The transition between Reflect and Respond phases labeled “dint/get queue(min); set ta” gets the minimum value in the queue and sets the time advance. The time advance shown on the other side of the transition arc is “time delay” indicating a variable time for the selected packet, this becomes the time advance for the Respond phase.

The Respond phase holds the packet for the time delay, usually equal to propagation delay of the device (the time it takes for the optical packet to enter one side and emit out the other). In this phase, we see both internal and external transitions. Since the time advance of the phase is not equal to zero, it is susceptible to external event interruption. As the isolator responds to both environmental and optical external events, it needs transitions for both. For the environmental external event, it is the same for the Passive phase: a check and return to the point of interruption shown by “dext ENV/update queue ta; check overtemp.” The difference here is the time spent in the phase, equal to *e*, subtracts from every packet in the queue and the optical packet transitioning the phase. This is shown by the “update queue ta” on the *dext* ENV transition arc.

If an optical packet arrives, the “dext OPT/update queue ta; insert(xi,ta)” transition arc leaves the Respond phase to the Reflect phase, as it follows the rule all reflections happen before anything else. Here we note the “update queue ta” happens again and the queue stores the new packets with the “insert(xi,ta)”. The component performs the Reflect phase for the new packets, then returns to the Respond phase.

But what happened to the packet that was in the Respond phase? This is where we use the *interruptRespond* state variable. This sets to true for an interrupted phase so the logic in the Reflect phase knows to return to the Respond phase without removing a new packet from the queue, with a new time advance equal to the time remaining for that interrupted packet. Back in the Respond phase, the packet remains for the remainder of the time delay, then the output function propagates the packet, as indicated by the “λ= propagation” notation under the name of the phase. The internal *dint* function has two choices: if the queue does not equal zero, it draws the minimum-time packet out of the queue for propagation; if the queue is empty, it advances to the Passive phase to await the next event.

Interrupted packets present a design difficulty, as each component changes a propagating packet during the output function, rather than during input. This decision ensures state changes from an environmental or control event affects the propagating packet, but also allows light entering the component behind the packet to affect it. The change to one packet is a minor effect when compared to the large amount of packets that travel through components and the simulation, on the order of $1x10^8$ packets per second. This design decision came from a discussion between the modeler, an optical physics SME, and the end user. Here we see the simulation design process in action where all parties involved agree to the model choices. Getting agreement on the necessary accuracy of the model is a way to increase the validity of the simulation.

The timescale chosen for the model allows for several design choices. Using the picosecond as the base time allows for discrete events to model continuous-time events, as mentioned earlier. This small period means that components only affect a few optical packets during each time unit (typical propagation time for a component is five picoseconds). For example, we chose to change the temperature of a device instantly. This is not true to the real system but the team decided this fast temperature change would affect relatively few packets and simplifies the design of the component. Once again, the users, modelers and optical SME accepted and agreed on this design abstraction. The timescale allows for the assumption that only one control and environmental message arrives to these ports at any given time, again simplifying the design.

The Appendix contains the complete DEVS pseudocode for the isolator.

C. DEVS Model of the Classical Pulse Generator

Since discussing the isolator design in-depth, we can look at the rest of the CPG components in comparison. The isolator design follows the base design for all optical components with a Passive phase, a Reflect phase and some form of a Respond phase. The isolator, polarizer, bandpass filter and the beamsplitter all share this common design. The beamsplitter differs by having four optical ports rather than two and splits an incoming optical into two propagating packets.

The laser and classical detector differ by having electric circuits in the device. They have an electric control port for control messages and a fourth phase to update the control logic within the device. The laser is unique as the only device to create optical pulses and the classical detector receives optical pulses and outputs a control message to the coupled controller. Response to optical pulses and environmental messages is the same in these two devices.

The PM fiber is a simple device with Passive and Respond phases because of the design decision that fiber does not create reflections when receiving optical pulses. It responds in the same manner as the isolator to environmental messages and its Respond phase works the same. The largest difference is the fiber deletes optical pulses if their power is below a specified limit.

The DEVS CPG model is a coupled model meaning is it comprised of atomic models. Fig. 12 shows the boundary of the CPG and the components. The CPG has environmental and control input and output ports on the left and optical input and output ports on the right. The CPG model has no functions or phases like other DEVS models, but because of DEVS composability and closure properties, it behaves as an atomic model. The inputs from external CPG ports connect to input ports on internal components and the output from one or more internal components connects to the CPG output ports. Internal components connect through component input and output ports.

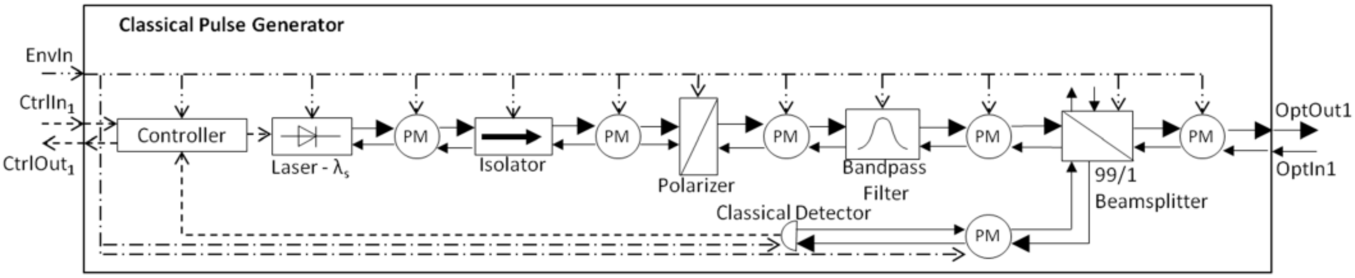


Fig 12. Conceptual DEVS architecture of the CPG.

The optical path starts with the laser and ends with the PM fiber linked to the CPG optical output. The primary travel direction for optical packets is indicated by the larger arrowhead in Fig. 12. Each of the optical components has a bidirectional optical connection shown by the arrows between each component. This is because the DEVS formalism decomposes a bidirectional connection into two separate unidirectional connections. The PM fiber connected to the beamsplitter output sends the optical packet to the CPG external port which sends the packet to the next subsystem. Each component has a one-way environmental port shown by the dotted-dashed lines. These ports connect to higher functions that send temperature updates to each component. Finally, there are connections from the CPG external and internal control ports to the controller and control port connections between the controller and laser and classical detector and controller.

The CPG controller is a simple abstraction of the electric circuits connecting devices in the CPG to the quantum controller. Its basic functions include receiving control messages from higher functions, passing messages to the laser and responding to status requests from higher functions. It receives the output control messages from the classical detector and stores detection data. It has DEVS pseudocode and functions as an atomic model.

Table II lists the messages from the quantum module controller to the CPG controller, Table III lists the messages from the classical detector to the controller, Table IV lists the messages sent by the CPG controller to the quantum module controller, and Table V lists the messages from the controller to the laser.

TABLE II

MESSAGES RECEIVED BY THE CPG CONTROLLER FROM THE QUANTUM MODULE CONTROLLER

Input Messages	Response
CPG_ENV	Set the internal CPG controller temperature
CPG_RESET	Resets the CPG controller and clears variables
CPG_STATUS_ REQUEST	Sends the CPG controller status and stored magnitude value
CPG_FIRE_ LASER	Sends a “Fire” command to the laser

TABLE III

MESSAGES RECEIVED BY THE CPG CONTROLLER FROM THE CLASSICAL DETECTOR

Input Messages	Response
CD_DETECTION	Store the magnitude of the detected laser pulse

TABLE IV

MESSAGES SENT FROM THE CPG CONTROLLER TO THE QUANTUM CONTROLLER

Output Messages	Content
CPG_ACK	Response to a Reset message
CPG_STATUS	Response to a Status Request message

TABLE V

MESSAGES SENT FROM THE CPG CONTROLLER TO THE LASER

Output Messages	Content
CPG_LASER_ FIRE	Command to fire the laser one time

The DEVS model of the CPG does not have the external, internal, output, confluence and time advance functions like an atomic model. Instead, the formalism specifies a set of all inputs, a set of all outputs, a list of internal model names, a list of the atomic models that comprise the coupled model, a list of external input and output connections and a list of internal component input and output connections. The Appendix contains the DEVS pseudocode for the CPG and its controller.

VI. DISCUSSION

A. *Discoveries Made During the Modeling Process*

Discoveries made during the conceptual modeling process fall into two categories: the modeling relation and the simulation relation. Recall the modeling relation is a measure of how close the model is to the source system and the simulation relation is a measure of how well the simulation executes the conceptual model instructions.

1) The Modeling Relation –

The original DEVS model failed to consider the change in attenuation in the EVOA happens over a time period determined by the device rate-of-change and the difference between the new and old attenuation values. Discussion with the optical SME uncovered this difference and lead to changing the DEVS model to an accuracy acceptable to the research team.

The problem with the EVOA identified the same condition with the polarization controller. In this case, it highlighted not only a necessary change in the components, but a change in the mathematical models for these components, as the necessary parameters were not in the original models.

Both of these rate-of-change problems necessitated a change to the MS4ME model that was not in line with pure DEVS theory, due to the current design of MS4ME. After consultation with the MS4 Systems modeling team, including Dr. Zeigler, we found an acceptable fix within MS4ME for the rate-of-change issue.

The DEVS modeler noticed the current QKD demonstration architecture uses only the polarization-independent version of the isolator, but other architectures use the polarization-dependent version. The optical SME realized the need to provide the mathematical model for the polarization-dependent version of the device for the DEVS model. The updated math model permitted the DEVS modeler to update the conceptual model to apply to both forms of the isolator.

2) The Simulation Relation –

Identifying the change to the EVOA DEVS model made the software modelers aware that these components in the proof-of-concept simulation had the same "instantaneous change" flaw. Using the updated conceptual model and input from the SME allowed for updates to the simulation code to capture the proper component rate-of-change.

Early in the DEVS modeling of pulsed light, we recognized the proof-of-concept simulation needed significant changes to handle continuous-wave light. The simulator models short-duration pulses using picosecond scales, but continuous-wave light cannot use the same abstraction method, requiring changes to the QKD simulator.

In the DEVS model, changes to the component during each optical packet propagation time affect the packet as it propagates through the component. Conversely, the QKD simulation receives a packet and schedules it for a future propagation. Since the component could change during the time before propagation, the scheduled optical pulse may not display proper behavior. This may require a change to how the QKD simulator handles optical packets. This speaks to the composability of DEVS and having well-defined behavior in the models.

B. *Did I Increase the Validity of qkdX?*

The concept of validity is not one of percentages or finite measurements. As defined earlier in this paper, validity is an expression of how difficult it is to differentiate between the model and source system outputs for the given experimental frame. Sargent suggests that "acceptance" of the model's accuracy for its intended purpose is the measure of conceptual model validity. Further he states that each submodel and the overall models need evaluation to determine if they are correct for the purpose of the model [19]. Two of the primary techniques for this are *face validation* and *traces*.

Face evaluation is where experts in the problem area evaluate the conceptual model to determine if it is correct and reasonable, usually by examining flowcharts or graphical models or a set of model equations [19]. In this research, we have research partners who are experts in quantum mechanics, QKD, physics and optics that constantly review the optical models and provide feedback and corrections as necessary.

Traces involve tracking entities through each atomic and coupled model determining if the logic and behavior associated with each is proper while maintaining necessary accuracy throughout [19]. The MS4ME simulator provided visual representations of the components as they transited through the models and produced detailed output to check the accuracy of the models during these tests.

As the models developed, the SMEs and research team provided feedback for correction, drawing each model closer to the expected system behavior, until each was deemed "acceptable," as discussed in Section III.a.2. Using the definitions of validity discussed earlier, this effort provided models that captured the required behavior and met the required accuracy, and so are considered "valid" with the understanding this validity only applies to the models built for the specific experimental frame. Any change to the experimental frame lessens or negates the validity of the models.

C. *Benefits of Using DEVS*

1) *Mathematically-proven Formalism for Creating a Conceptual Model*

With the understanding that a DES is a finite state machine with a set of triples of inputs, states, and outputs to describe each state; DEVS captures these in a logical manner and provides a formal language to describe the conceptual model. DEVS uses set theory to prove its applicability to DES models [43].

2) *Tool-independent Form of the Model*

One of the central ideas of DEVS is the model is the bridge between the source system and the simulation. There is no direct connection between the source system and simulation, meaning the conceptual model created using DEVS is applicable to any simulation, if the simulation is capable of implementing the DEVS-specified behavior. A DEVS conceptual model is independent of a specific simulation.

3) *Closure Under Coupling Within the Formalism*

DEVS allows the modeler to build hierarchal systems from smaller subsystems and components. This property, also called composability, ensures DEVS models connect in any manner, produce expected behavior and the coupled models are behaviorally equivalent to atomic models. Together with tool-independence, these properties allow for using repositories of models in a "mix and match" environment.

4) *Canonical Understanding of the Model Behavior*

DEVS forces the modeler to carefully consider all facets of desired behavior within the model, including all inputs, outputs, and timing segments. This greatly reduces or eliminates emergent or unexpected behavior. By delving into the source system and creating a set of atomic models not further decomposable, the modeler creates a deep understanding of the system usable in any modeling simulation.

5) *Well-defined Behavior in the Conceptual Model*

Having a canonical understanding enables the modeler create a set of rules for each atomic and coupled model, creating well-defined behaviors. This behavior ensures the model never enters a situation where it moves into a passive state without a way to become active (unless this is a behavior the modeler desires). DEVS defines the conditions necessary to ensure this well-defined behavior.

D. *Limitations of Using DEVS*

1) *Applying DEVS to Complex Components*

Much of the written material available for DEVS in textbooks and papers provides only simplistic examples. Even the more complex examples available through RTSync provided little guidance to model the optical components. This complexity led this researcher to contact Dr. Sarjoughian, co-director of ACIMS, for advice on using DEVS for our unique, innovative models. He provided suggestions on how to use DEVS to model the timing issues at the picosecond scale and capturing wave and particle behavior.

It can be difficult to verify the DEVS pseudocode to the source system behavior, especially true when modeling predicted or notional systems. Since our demonstration QKD system is built from real optical components but in a notional architecture, we had the difficulty of verifying the component behavior. Our solution to this problem greatly increased the DEVS work by necessitating programing the pseudocode twice, once into MS4ME, and then into the selected qkdX simulator.

2) *DEVS and Processes and Information Flows*

DES models are increasing in complexity and being used in new information-centered fields. DEVS was not created for these types of problems and has difficulty expressing processes and information flows within the formalism. Heretofore, the solution has been to create subsets of DEVS to handle the specific problem. This is leading to a fragmentation of the formalism and makes it hard to determine which form of DEVS is appropriate for the modeling problem.

3) *Not Visual Without Using a DEVS Simulator*

DEVS is a set of language rules to formally describe a problem. There is no visual component to DEVS unless the modeler uses a DEVS-compliant simulation program. While very useful for being tool-independent, this requires the modeler to use that particular simulator's functions, which may not necessarily conform completely to DEVS, as seen with the EVOA timing issue and MS4ME.

VII. Conclusions

The intent of this research is to show how the DEVS formalism could increase the validity of the qkdX simulation for its designed purpose. The question of what is sufficiently accurate is the question of validity, as noted by Zeigler and Robinson. When used properly, DEVS increases the validity of simulation, for its intended purpose. No simulation is valid for all purposes, so it is important understand Zeigler's idea of the experimental frame so to carefully limit discussing validity to an intended purpose.

To test our hypothesis, we created DEVS models for atomic components in the Alice CPG and programmed them into a DEVS-compliant simulator to check their correctness. The SMEs and the research team reviewed the results (face validity) and checked the output values against the required accuracies as events moved through the models (tracing). After correction, the atomic models were used in a coupled model, the CPG, demonstrating the hierarchal properties of DEVS.

Using DEVS allowed the team to refine the qkdX simulation, correcting several errors and aiding the research team in recognizing missing behaviors within the simulation. DEVS increased the validity of qkdX optical pathway by aiding the team in making qkdX simulation behavior closer to the source system behavior, showing DEVS can be used to increase validity by creating optical component models fit for the purposes of the simulation and acceptable to the community of developers and users.

While having a large learning curve, DEVS proved to be a valuable tool for our research.

VIII. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] S. Singh, *The Code Book: The Secret History of Codes and Code-Breaking.* London, England: Fourth Estate, 1999, pp. ix.

[2] E. B. Barker, W. C. Barker and A. Lee. (2005, Dec.) Guideline for Implementing Cryptography in the Federal Government. NIST, Gaithersburg, MD. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-21-1/sp800-21-1_Dec2005.pdf

[3] C. E. Shannon. (1946, Sep). Communication Theory of Secrecy Systems. Bell Labs. USA. [Online]. Available: http://dm.ing.unibs.it/giuzzi/corsi/Support/papers-cryptography/Communication_Theory_of_Secrecy_Systems.pdf

[4] C. E. Shannon. (1948, Oct). A Mathematical Theory of Communication. *The Bell System Technical Journal. 27*, pp. 379-423. Available: http://www.inf.ed.ac.uk/teaching/courses/com/handouts/extra/shannon-1948.pdf

[5] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. New York: John Wiley & Sons, 1995, pp. 151-153.

[6] M. R. Grimaila, J. D. Morris and D. Hodson, "Quantum Key Distribution, A revolutionary security technology," *The ISSA Journal,* vol. 27, pp. 20-27, Jun. 2012.

[7] Austrian Institute of Technology. QKD Software. [Online]. Available: https://sqt.ait.ac.at/software/projects/qkd-software

[8] J. D. Morris, D. D. Hodson, M. R. Grimaila, D. R. Jacques and G. Baumgartner. (2014, Mar.) Towards the modeling and simulation of quantum key distribution systems. *International Journal of Emerging Technology and Advanced Engineering.* [Online]. *4(2)*. Available: http://www.ijetae.com/files/Volume4Issue2/IJETAE_0214_143.pdf

[9] B. P. Zeigler, G. Ball, H. Cho, J. Lee and H. Sarjoughian. (1999) Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions. Presented at the Simulation Interoperation Workshop. [Online]. Available: http://acims.asu.edu/wp-content/uploads/2012/02/SIWDEVSImplemHLARTI.pdf.

[10] B. P. Zeigler, H. S. Song, T. G. Kim and H. Praehofer. (1995). *DEVS framework for modelling, simulation, analysis, and design of hybrid systems* [Online]. Available:http://www.researchgate.net/publication/2684908_DEVS_Framework_for_Modelling_Simulation_Analysis_and_Design_of_Hybrid_Systems/file/72e7e5215bfca6eaeb.pdf.

[11] G. A. Wainer and N. Giambiasi. (2001) Application of the cell-DEVS paradigm for cell spaces modelling and simulation. *Simulation.76(1),* pp. 22-39. Available: http://cell-devs.sce.carleton.ca/papers/Simulation01.pdf

[12] B. P. Zeigler and D. Kim. (1999, Dec.) Distributed supply chain simulation in a DEVS/CORBA execution environment. Presented at the 31st Winter Simulation Conference. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.9681&rep=rep1&type=pdf

[13] S. Mittal, J. L. Risco and B. P. Zeigler. (2007, Jul.) DEVS-based simulation web services for net-centric T&E. Presented at the 2007 Summer Computer Simulation Conference. [Online]. Available: http://cell-devs.sce.carleton.ca/citations/SC208_Mittal.pdf

[14] L. Ntaimo, B. P. Zeigler, M. J. Vasconcelos and B. Khargharia. (2004, Oct.). Forest fire spread and suppression in DEVS. *Simulation.* [Online]. *80(10),* pp. 479-500. Available: http://ise.tamu.edu/people/faculty/Ntaimo/personal_web/Papers/NtaimoSIM012004.pdf

[15] G. Wainer. (2006, Oct.) Applying cell-DEVS methodology for modeling the environment. *Simulation.* [Online]. *82(10),* pp. 635-660. Available: http://cell-devs.sce.carleton.ca/publications/2006/Wai06/Environment635.pdf

[16] H. B. Gunay, L. O'Brien, R. Goldstein, S. Breslav and A. Khan. (2013, Apr.) Development of discrete event system specification (DEVS) building performance models for building energy design. Presented at the Symposium on Simulation for Architecture & Urban Design. [Online]. Available:http://www.autodeskresearch.com/pdf/46_Final_Paper.pdf.

[17] O. Balci. (1995, Dec.). Principles and techniques of simulation validation, verification, and testing. Presented at the 1995 Winter Simulation Conference. [Online]. Available: http://www.informs-sim.org/wsc95papers/1995_0021.pdf

[18] O. Balci. (1997, Dec.). Verification validation and accreditation of simulation models. Presented at the 29th Winter Simulation Conference. [Online]. Available: http://www.informs-sim.org/wsc97papers/0135.PDF

[19] R. G. Sargent. (2005, Dec.) Verification and validation of simulation models. Presented at the 37th Winter Simulation Conference. [Online]. Available: http://student.telum.ru/images/6/66/Sargent_VV_2010.pdf

[20] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 75-76.

[21] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 142.

[22] B. P. Zeigler, H. Sarjoughian, *Guide to Modeling and Simulation of Systems of Systems*. London England: Springer-Verlag, 2013, pp. 24.

[23] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 149-152.

[24] O. Balci, J. D. Arthur and W. F. Ormsby, “Achieving reusability and composability with a simulation conceptual model,” *Journal of Simulation*, vol. 5, no. 3, pp. 157-165, Mar. 2011.

[25] M. Hofmann, J. Palii and G. Mihelcic, “Epistemic and normative aspects of ontologies in modelling and simulation,” *Journal of Simulation* vol. 5, no. 3, pp. 135-146, Jul. 2011.

[26] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 25-26.

[27] A. Tolk, S. Y. Diallo, J. J. Padilla and H. Herencia-Zapana, “Reference modelling in support of M&S—foundations and applications,” *Journal of Simulation* vol. 7 no. 2, pp. 69-82, Feb. 2013.

[28] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 76.

[29] B. P. Zeigler and H. S. Sarjoughian. (2005). Approach and techniques for building component-based simulation models. [Online]. Available: http://acims.asu.edu/wp-content/uploads/2012/02/iitsec.ppt

[30] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 149-152.

[31] S. Robinson, “The future's bright the future's… Conceptual modelling for simulation!,” *Journal of Simulation*, vol. 1, no. 3, pp. 149-152, 2007.

[32] S. Robinson, *Simulation: The Practice of Model Development and Use.* Chichester, England: John Wiley & Sons, 2004, pp. 65.

[33] S. Robinson, *Simulation: The Practice of Model Development and Use.* Chichester, England: John Wiley & Sons, 2004, pp. 77-78.

[34] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 31.

[35] C. Szabo and Y. M. Teo. (2012, Jun.). An analysis of the cost of validating semantic composability. *Journal of Simulation*. [Online]. *6(3)*, pp. 152-163. Available: http://www.comp.nus.edu.sg/~teoym/pub/12/jos201211a.pdf

[36] A. Vargas, “OMNeT++,” in *Modeling and Tools for Network Simulation*. Berlin, Germany: Springer-Verlag, 2010, pp. 35-59.

[37] Wolfram. Wolfram Mathematica. [Online]. Available: http://www.wolfram.com/mathematica/

[38] MS4 Systems. MS4ME. [Online]. Available: http://www.ms4systems.com/pages/ms4me.php

[39] Arizona State University. Arizona Center for Integrative Modeling and Simulation. [Online]. Available: http://acims.asu.edu/

[40] C. H. Bennett and G. Brassard. (1984). Quantum cryptography: Public Key Distribution and Coin Tossing. Presented at the IEEE International Conference on Computers, Systems and Signal Processing. [Online] Available: http://www.cs.ucsb.edu/~chong/290N-W06/BB84.pdf

[41] ID Quantique. Quantus Optical Random Number Generator. [Online]. Available: http://www.idquantique.com/component/content/article.html?id=9

[42] B. P. Zeigler, H. Praehofer and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems,* 2nd ed. San Diego: Academic Press, 2000, pp. 89-93.

[43] B. P. Zeigler, *Theory of Modeling and Simulation*. New York: John Wiley & Sons, 1976, pp. 293-301.

APPENDIX

A. Isolator DEVS Pseudocode

$\boldsymbol{DEVS_{Isolator} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
("reflect", 0, store, temperature, overtemp, overpower,interruptRespond,  queue.x1..xn)
    if phase = "passive" and $p \in$ {"OptIn$_1$", "OptIn$_2$"}
        for messagebag != null
           current = messagebag_first()
            if current.value.power > damaged.power
            overpower =  "Y"
         insert_event_q(current)
         remove_event_m(current)
     queue.current = queue_first(queue)
       reflect = (queue.current.p), calcReflected(queue.current.v))
       mark_reflected(queue.current)
       interruptRespond = "N"

("reflect", 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "respond" and $p \in$ {"OptIn$_1$", "OptIn$_2$"}
        update_delay(queue)
        for messagebag != null
          current = messagebag_first()
          if current.value.power > damaged.power
          overpower =  "Y"
       insert_event_q(current)
        remove_event_m(current)
     queue.current = queue_need_reflected()
     reflect = (queue.current.p), calcReflected(queue.current.v))
      mark_reflected(queue.current)
     interruptRespond= "Y"
    timeLeftRespond = timeLeftRespond - e

("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "passive" and p = "EnvIn"
     temperature = messagebag.temperature
     if temperature > damage.temp
        overtemp = "Y"

("respond", time.delay,  store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "respond" and p = "EnvIn"
      update_delay(queue)
      timeLeftRespond = time.delay- e
      temperature = messagebag.temperature
      if temperature > damage.temp
        overtemp = "Y"
      time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue, e, (($p_i$,$v_i$),…. ($p_n$,$v_n$))) =

("reflect", 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
    if phase = "reflect" and need.reflect != null
     need.reflect = queue_need_reflected()
     current = need.reflect
      reflect = (current.p), calcReflected(current.v))
      mark_reflected(current)

("respond", time.delay,  store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = "reflect" and need.reflect = null
     need.reflect = queue_need_reflected()
     if interruptRespond = "N"
        current = queue_min()
        time.delay = current.time.delay
        if InPort = "OptIn$_1$"
          outputPulse = calcForward(current.v, temperature, overtemp, peakpwr, overpwr)
          outputPort = “OptOut$_2$”
        if InPort = “OptIn$_2$”
          outputPulse = calcReverse(current.v, temperature, overtemp, peakpwr, overpwr)
          outputPort = “OptOut$_1$”
      timeLeftRespond = propagation delay
    else
      time.delay = timeLeftRespond

  (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if InPort = “OptIn$_1$”
        outputPulse = calcForward(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_2$”
      if InPort = “OptIn$_2$”
        outputPulse = calcReverse(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_1$”
      interruptRespond= “N”

  (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size  = 0
      size= queue_size()
```

Confluence Function:

$$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$$

Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower) =
    (reflect.p, reflect.v)
        if phase = “reflect”

    (outputPort, outputPulse)
      if phase = “propagate”

    $\emptyset$ (null output)
    otherwise;
```

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower, interruptRespond, queue) = \sigma;$

B. Classical Pulse Generator Controller DEVS Pseudocode

$\boldsymbol{DEVS_{CPGcontroller} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$

`

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, lastCDPower, e, (($p_i$,$v_i$),…. ($p_n$,$v_n$))) =
(“respond”, 0, store, temperature, overtemp, overpower, lastCDPower)
    if phase = “passive” and p = “CtrlIn$_1$”
    ctrlOutput = ctrlMsg(store)
    if ctrlMsg.status = “init” or “get status”
      outputPort = “CtrlOut$_1$”
    if ctrlMsg.status = “fire laser”
      outputPort = “CtrlOut$_2$”

(“passive”, 0, store, temperature, overtemp, overpower, lastCDPower)
    if phase = “passive” and p = “CtrlIn$_2$”
     lastCDPower = messagebag.magnitude

(“passive”, $\infty$, store, temperature, overtemp, overpower, lastCDPower)
    if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

(phase, $\sigma$ – e, store, temperature, overtemp, overpower, lastCDPower)
otherwise;
```

Internal Transition Function:

δ_{int}(*phase*, *σ, store, temperature*, *overtemp*, *overpower*, *lastCDPower*) =
 (“passive”, ∞, *store, temperature*, *overtemp*, *overpower*, *lastCDPower*)
  if *phase* = “respond”

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:

λ(*phase*, *σ*, *store*, *temperature*, *overtemp*, *overpower*, *lastCDPower*) =
 (*output.port*, *output.pulse*)
  if phase = “respond”

 (*outputPort*, *ctrlOutput*)
  if phase = “respond”

 ∅ (null output)
 otherwise;

Time advance Function:

ta(*phase*, *σ*, *store*, *temperature*, *overtemp*, *overpower*, *lastCDPower*) = *σ*;

C. Classical Pulse Generator Coupled Model Pseudocode

***DEVS$_{CPG}$* = (*X, Y, D,* {*M$_d$* | d ∈ *D*}, *EIC, EOC, IC*)**

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$”, “OptIn$_1$”, “OptIn$_3$”, “OptIn$_4$”, “EnvIn”}
X = {(“CtrlIn$_1$”, *v*), (“CtrlIn$_2$”, *v*), (“OptIn$_1$”, *v*), (“OptIn$_3$”, *v*), (“OptIn$_4$”, *v*), (“EnvIn”, *v*) |*v* ∈ *V*}

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”, “OptOut$_1$”, “OptOut$_3$”, “OptOut$_4$”}
Y = {(“CtrlOut$_1$”, *v*), (“CtrlOut$_2$”, *v*), (“OptOut$_1$”, *v*), (“OptOut$_3$”, *v*), (“OptOut$_4$”, *v*) |*v* ∈ *V*}

D = {controller, laser, isolator, polarizer, bandpass, beamsplitter, classicaldetector, PMfiber}
$M_d = M_{controller}, M_{laser}, M_{isolator}, M_{polarizer}, M_{bandpass}, M_{beamsplitter}, M_{classicaldetector}, M_{PMfiber}$

EIC = {((*N*, “CtrlIn$_1$”),(controller, “CtrlIn$_1$”)), ((*N*, “EnvIn”),(controller, “EnvIn”)), ((*N*, “EnvIn”),(laser, “EnvIn”)), ((*N*, “EnvIn”),(isolator, “EnvIn”)), ((*N*, “EnvIn”),(polarizer, “EnvIn”)), ((*N*, “EnvIn”),(bandpass, “EnvIn”)), ((*N*, “EnvIn”),(beamsplitter, “EnvIn”)), ((*N*, “EnvIn”),(classicaldetector, “EnvIn”)), ((*N*, “EnvIn”),(PMfiber, “EnvIn”)), {((*N*, “OptIn$_1$”),(PMfiber, “OptIn$_2$”))}

EOC = {((PMfiber, “OptOut$_2$”),(*N*, “OptOut$_1$”)), ((controller, “CtrlOut$_1$”),(*N*, “CtrlOut$_1$”))}
IC = {((controller, “CtrlOut$_2$”), (laser, “CtrlIn”)), ((laser, “OptOut$_1$”),(PMfiber, “OptIn$_1$”)), ((PMfiber, “OptOut$_2$”), (isolator, “OptIn$_1$”)), ((isolator “OptOut$_2$”), (PMfiber, “OptIn$_1$”)), ((PMfiber, “OptOut$_2$”), (polarizer, “OptIn$_1$”)), ((polarizer “OptOut$_2$”), (PMfiber, “OptIn$_1$”)), ((PMfiber, “OptOut$_2$”), (bandpass, “OptIn$_1$”)), ((bandpass “OptOut$_2$”), (PMfiber, “OptIn$_1$”)), ((PMfiber, “OptOut$_2$”), (beamsplitter, “OptIn$_1$”)), ((beamsplitter, “OptOut$_4$”),(PMfiber, “OptIn$_1$”)), ((beamsplitter, “OptOut$_3$”),(PMfiber, “OptIn$_2$”)), ((PMfiber, “OptOut$_1$”),(classicaldetector, “OptIn$_1$”)), ((classicaldetector, “CtrlOut”),(controller, “CtrlIn$_2$”)), ((classicaldetector, “OptOut$_1$”), (PMfiber, “OptIn$_1$”)), ((PMfiber, “OptOut$_2$”), (beamsplitter, “OptIn$_3$”)), ((PMfiber, “OptOut$_1$”), (beamsplitter, “OptIn$_4$”)), ((beamsplitter, “OptOut$_1$”),(PMfiber, “OptIn$_2$”)), ((PMfiber, “OptOut$_1$”), (bandpass, “OptIn$_2$”)), ((bandpass, “OptOut$_1$”),(PMfiber, “OptIn$_2$”)), ((PMfiber, “OptOut$_1$”), (polarizer, “OptIn$_2$”)), ((polarizer, “OptOut$_1$”),(PMfiber, “OptIn$_2$”)), ((PMfiber, “OptOut$_1$”), (isolator, “OptIn$_2$”)), ((isolator, “OptOut$_1$”),(PMfiber, “OptIn$_2$”)), ((PMfiber, “OptOut$_1$”), (laser, “OptIn$_2$”))}

8. Simulation Results and Analysis

The purpose of this chapter is to present the results of the research by discussing the qkdX QKD simulation framework, the model creation process, providing samples of simulation output and analyzing the output. Several data tables provide summaries for the components and coupled submodules.

8.1 *The QKD Simulation Framework (qkdX)*

The AFIT QKD research team developed a QKD simulation framework (qkdX) to enable efficient modeling of QKD systems for performance analysis and characterization. Design features of qkdX include: a hybrid discrete-continuous modeling approach to more accurately capture quantum effects; a modular design to allow quick and efficient changes to the system under study; parameterized components allowing for multiple varying instances; a composable system allowing for hierarchal construction of complex systems from simple components.

This capability allows users (e.g., engineers or analysts) to more quickly model QKD systems, enabling security and performance analysis, including:

- Model and analyze competing QKD implementations (e.g., variations in hardware components or software processes)
- Increase understanding of the security-performance design and implementation trade space for realized QKD systems
- Determine the impact of non-idealities and practical engineering limitations in QKD architectures
- Identify interactions between physical (quantum phenomenon, temperature, and disturbances) and system-level interactions (hardware designs, software implementations, and protocols)
- Propose and assess new QKD implementations or protocols

- Study the security implications of different protocol modifications and system architectures
- Model performance characteristics of free-space and space-based QKD systems
- Maximize research development efforts to improve implementations (e.g., should one invest research capital in on-demand single-photon sources or improved single photon detectors?)

The framework is designed with considerations to support multiple qubit encoding schemes (i.e., polarization-based, phase-based, and entanglement), multiple protocols (e.g., BB84, SARG04, E92), and various QKD applications (e.g., buried optical fiber, terrestrial directional free-space optical link, and multiplexed transmissions). Initially, the framework was used to model a notional polarization-based, prepare-and-measure BB84 terrestrial fiber QKD system. This research constructed and tested the conceptual models for the optical path components necessary for BB84 system.

8.2 *Component Modeling*

The process for creating each coupled and atomic model for the reference architecture consisted of multiple steps. Appendix B has a detailed description of model creation and testing but from an overview perspective, the process creating and testing the optical models consisted of:

1. Optical SME created a mathematical model of the component
2. Use the mathematical model to create a component conceptual model
3. Create DEVS pseudocode to capture the behavior and timing aspects
4. Create MS4ME models using the DEVS pseudocode
5. Test the MS4ME models to ensure proper behavior and timing
6. Share results with optical SME throughout the steps to check models

Each component model started with a mathematical model from the optical SME. A description of the component based on the mathematical model, commercial data sheets and academic literature provided the basic understanding needed for the conceptual model. The conceptual model and terse uses cases for the component lead to a series of English-language rules describing the high-level behavior of the component and these were captured graphically in a phase transition diagram. Event-trace diagrams, in the form of state tables, provided a written version of the phase changes shown in the phase transition diagram.

DEVS pseudocode captured the identified component behavior and timing using the preceding diagrams, rules and use cases. The pseudocode was programmed into the DEVS-compliant MS4ME simulator with the output captured for later evaluation. After manually checking the component output against the mathematical model and expected behavior in the DEVS pseudocode, the MS4ME models were completed after multiple iterations of testing and correction, and then used to construct the coupled submodules in MS4ME.Table 2 lists the appendix containing the related documents for each component and coupled submodule.

Table 2. *List of Modeled Components and Submodules.*

Appendix #	**Component**	**Appendix #**	**Component**
D	Bandpass Filter	P	Pulse Modulator
E	Beamsplitter	Q	Polarizing Beamsplitter
F	Circulator	R	SM Fiber
G	Optical Photodiode (Classical Detector)	S	Optical Switch
H	EVOA	T	WDM
I	Fixed Attenuator	U	CPG Module
J	Half-wave Plate	V	PM Module
K	In-line Polarizer	W	DSG Module
L	Isolator	X	CTQ Module
M	Laser	Y	OSL Module
N	PM Fiber	Z	TPG Module
O	Polarization Controller	AA	OPM Module

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests the component behavior when it is not active and awaiting input and tests the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature. All identified errors were corrected and the component retested until it functioned properly for each test case. Once the component completed testing, the component documentation and results went to the optical SME for review.

Table 3 summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework (see chapter 7 for a discussion on this framework).

Table 3. *Summary of Component Behavior Testing.*

	Passive Phase								Respond Phase						
	total tests	optical ports	ctrl port	env port	single port	multiple port	math tests		total tests	optical ports	ctrl port	env port	single port	multiple port	math tests
Bandpass Filter	21	20	0	13	5	16	4		21	21	0	13	4	17	0
Beamsplitter	33	28	0	21	9	24	6		33	33	0	21	8	25	0
Circulator	27	26	0	17	6	21	6		27	27	0	17	6	21	0
Classical Detector	21	14	14	13	5	16	7		21	21	14	13	1	20	0
EVOA	30	24	13	18	6	24	7		22	22	9	11	2	19	0
Fixed Attenuator	21	20	0	13	5	16	7		21	21	0	13	4	17	0
Half-wave Plate	21	20	0	13	5	16	8		21	21	0	13	4	17	0
In-line Polarizer	49	20	0	13	5	16	7		21	21	0	13	4	17	0
Isolator	49	20	0	13	5	16	7		21	21	0	13	4	17	0
Laser	21	14	14	13	5	16	7		21	21	15	13	2	19	0
Optical Switch	35	29	14	19	7	28	8		27	27	10	12	2	25	0
PM Fiber	21	20	0	13	3	18	7		21	21	0	13	4	17	0
Polarization Controller	30	24	13	18	6	24	8		22	22	9	11	2	20	0
Polarization Modulator	30	24	13	18	6	24	8		22	22	9	11	2	20	0
Polarizing Beamsplitter	33	28	0	21	9	24	7		33	33	0	21	8	25	0
SM Fiber	21	20	0	13	3	18	7		21	21	0	13	4	17	0
Wave Division Multiplexer	27	26	0	17	4	23	7		27	27	0	17	6	21	0

8.3 *Coupled Submodule Modeling*

The process for creating each coupled submodule used the same process in creating the components with one major difference. Each coupled model uses the prototypical demonstration architecture as a starting basis (see chapter 7), rather than a mathematical model. Under DEVS, a coupled model does not have its own logic or behavior, but rather serves as a repository of atomic models (see the DEVS code for the coupled models in each of the appendices U-AA). Some coupled submodules have a simple controller component that represents the logic necessary to control any opto-electrical devices within the submodule. Each of these controllers has typical DEVS atomic behavior.

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The control message passed to each coupled submodule depended on the internal components.

1. CPG submodule – control message fires signal laser
2. PM submodule – control message changes polarization of polarization controller
3. DSG submodule – control message changes attenuation of EVOA
4. CTQ submodule – control message changes attenuation of EVOA
5. OSL submodule – no control message to change internal settings
6. TPG submodule – control message fires timing laser
7. OPM submodule – control message changes optical switch position

The exceptions for the test cases were the CPG and the OSL. The CPG did not have a test case to inject an optical packet as the CPG is the module that creates the optical pulse. The OSL had one less control message as its “controlled” device is the classical detector, which only sends data to the controller, and does not accept input control message

These test cases led to iterations of testing and correction (see appendix B for detailed descriptions). All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling. See Table 4 for a summary of these tests.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

		Test Type		
	total tests	optical port	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

8.3.1 *CPG Testing*

In the following figures, Figure 2 is the CPG architecture diagram, Figure 3 is the high-level test frame and Figure 4 shows the exploded view of the CPG submodule within the high-level test frame. Notice there is a “cpgcontroller” that receives messages from outside the CPG. This is a simple representation of the logic circuits necessary to operate this module. In these conceptual models, the controller reacts to the appropriate message types for the opto-electrical components connected to the controller. For example, the CPG controller accepts messages for the laser and the classical detector. Similarly, each coupled model has a controller specific for its needs.

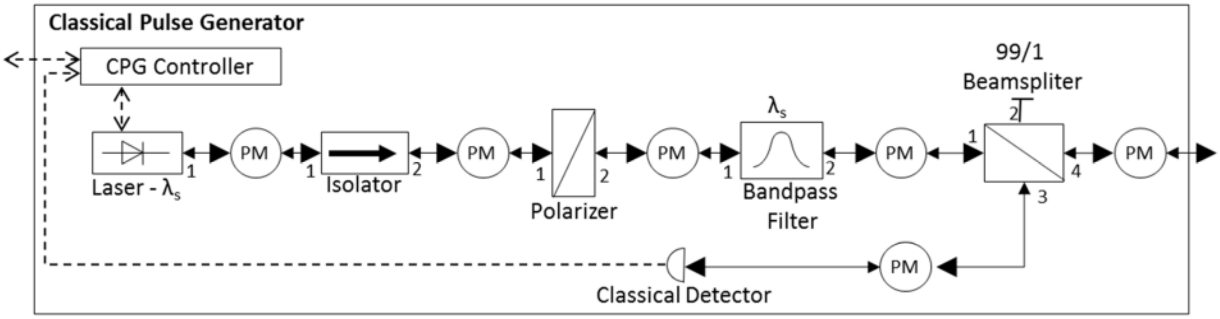


Figure 2. CPG architecture.

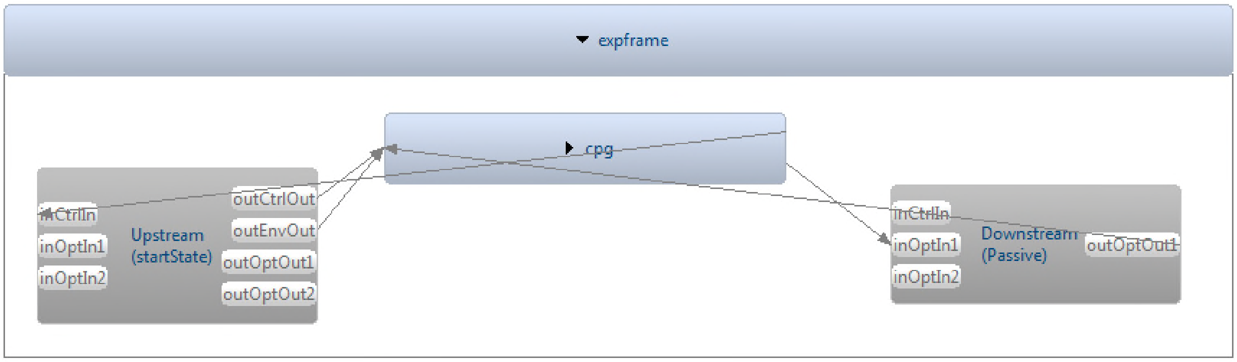


Figure 3. CPG test frame.

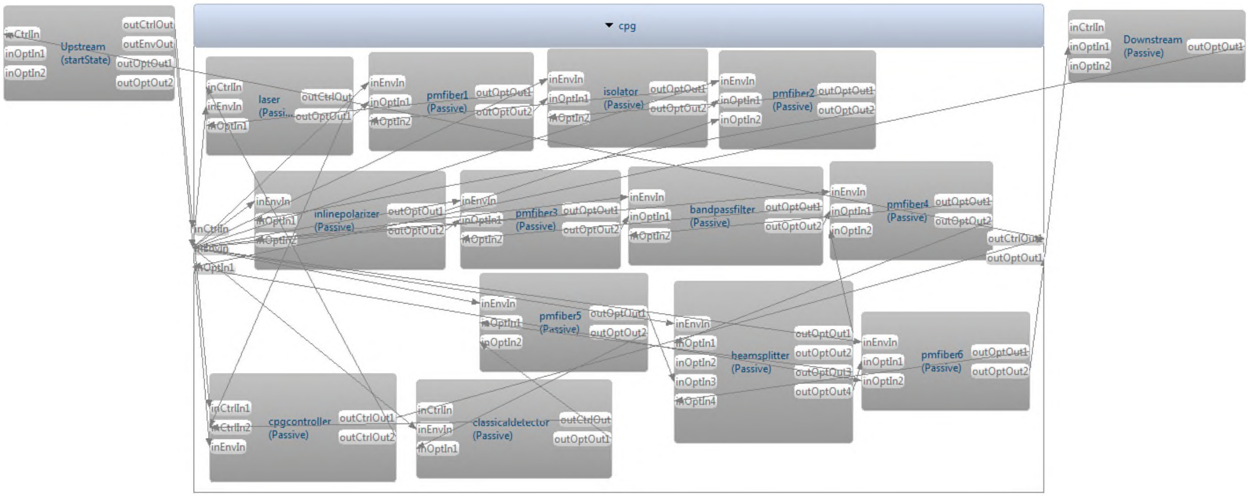


Figure 4. CPG coupled submodule components.

The testing construct for each coupled module is the same. There is test component called “Upstream” that injects messages into the submodule under test and a testing component called “Downstream” that captures all output from the submodule.

Figure 3 shows the *expframe* module with the two test components and the CPG submodule, each labeled with its current phase and ports. As noted earlier, Upstream only has output ports (labeled with “out”) connected to the CPG input ports (labeled with “in”), and all of the CPG output ports are connected to Downstream’s input ports. Since the CPG does not output environmental (labeled “env”) messages, there are is no environmental input port in Downstream. Each optical component and submodule listed in Table 2 has a similar testing construct built in MS4ME.

The code necessary for building coupled models much simpler than the code for the components. The coupled model code specifies external components (Upstream, Downstream) and external inputs and outputs for the CPG module. Additionally, the code lists all internal components and the internal connections between them. Finally, the connections from the CPG to internal components are defined. For a coupled model, there are no phases or transitions defined, as the ‘state’ of the coupled model is the sum of all current states of all internal components. Appendix U describes the CPG in detail and each coupled module appendix (U-AA) has the DEVS code for the controller and the coupled module.

Once constructed, each coupled model was tested by injecting the proper message packet into the submodule. Every coupled model, except the CPG, has input optical, environmental and control ports. The CPG has no input optical port, as the laser within the CPG generates optical pulses for the Alice subsystem and its primary input is a control message to fire the laser. Table 5 summarizes the test cases for the CPG submodule.

Table 5. *Example test case timing chart - CPG.*

	Inject Ports				Running Totals		
Case	**Opt1**	**Ctrl**	**Env**	**Timing**	**opt #**	**env #**	**ctrl #**
1	0	1	0	single	0	0	1
2	0	1	0	single	0	0	2
3	0	0	1	single	0	1	2
4	0	1	0	single	0	1	3
totals	0	3	1				

8.3.2 *CPG MS4ME Output*

This section contains the output from MS4ME while running the CPG submodule. This is the output from Test Case 1 (highlighted in Table 5) and starts at simulation time 10. Major events described in this paragraph are highlighted in yellow in the following MS4ME output. Case 1 starts with Upstream having an output event at time 10, sending the message to the CPG which seamlessly passes the message inside the submodule to the CPG controller. The controller receives a LaserMsg with id:1 on port CtrlIn1 and starts the Passive external event. The controller sees it has received a laser fire message and moves to the Respond phase. The controller notes that the status number of the message is 6, the stored power value of the last classical detection is zero (this is sent from the classical detector when it has a detection) and prepares a response message and sends it out port CtlrOut2, ending the Response phase.

The response message goes to the laser, which receives the message on port CtrlIn and starts the Passive CtrlIn event. The laser decodes the message, moves to the “ON” setting and prepares a fire control message for port CtrlOut. The laser moves to the Update Laser phase where it notes the laserpower variable is “true,” indicating the laser is on. The laser begins preparing an optical pulse for output on OptOut1 (the only optical output port in the laser) and notes it is moving to the Create Pulse phase. Once in the

Create Pulse phase, the laser outputs the generated optical pulse on port OptOut1 and ends the Create Pulse phase. Finally, the next component in line from the laser, the pmfiber1, receives the optical pulse at simulation time 16. This output shows how the CPG, the CPG controller, and the CPG internal laser reacts to a “fire laser” message to generate optical pulses at the beginning of the optical path within Alice.

```
[10.0, 2:13:36.632] SimViewer_expframe: Simulation step started
[10.0, 2:13:36.632] Upstream - output event for Case1 - time Elapsed: NA getTimeAdvance: 10.0 clock: 10.0
[10.0, 2:13:36.632] SimViewer_expframe.expframe.expframe.Upstream: Internal transition
[10.0, 2:13:36.632] SimViewer_expframe.expframe.expframe.Upstream: Internal transition from Case1
[10.0, 2:13:36.632] SimViewer_expframe.expframe.expframe.Upstream: Holding in phase Case1_1 for time 1.0
[10.0, 2:13:36.632] Upstream - internal event for Case1  - time Elapsed: NA getTimeAdvance: 1.0 clock: 10.0
[10.0, 2:13:36.632] Upstream - internal event for Case1 - time Elapsed: NA getTimeAdvance: 10.0 clock: 10.0
[10.0, 2:13:36.632] SimViewer_expframe: Simulation step finished
[11.0, 2:13:36.773] SimViewer_expframe: Simulation step started
[11.0, 2:13:36.788] Upstream - output event for Case1_1 - time Elapsed: NA getTimeAdvance: 1.0 clock: 11.0
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.Upstream: Internal transition
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.Upstream: Internal transition from Case1_1
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.Upstream: Holding in phase Wait1 for time 49.0
[11.0, 2:13:36.788] Upstream - internal event for Case1_1 - time Elapsed: NA getTimeAdvance: 49.0 clock: 11.0
[11.0, 2:13:36.788] Upstream - internal event for Case1_1 - time Elapsed: NA getTimeAdvance: 1.0 clock: 11.0
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Received messages [inCtrlIn1: LaserMsg
                id: 1
                name: Case 1 Control Message 1
                status: 6
                magnitude: 0.0]
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: External transition
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Respond for time 0.0
[11.0, 2:13:36.788] @@************* CPG CONTROLLER - START PASSIVE CTRLIN1 EXTERNAL
EVENT********************************************************************
[11.0, 2:13:36.788] CPG Controller - external event for Passive CtrlIn with CtrlIn - time Elapsed: 11.0 getTimeAdvance: 0.0 clock:
11.0
[11.0, 2:13:36.788] CPG Controller received a laser fire message
[11.0, 2:13:36.788] CPG Controller - Preparing laser fire control message for: Case 1 Control Message 1 to port CtrlOut2
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Respond for time 0.0
[11.0, 2:13:36.788] @@************* CPG CONTROLLER - END PASSIVE CTRLIN1 EXTERNAL
EVENT********************************************************************
[11.0, 2:13:36.944] SimViewer_expframe: Simulation step finished
[11.0, 2:13:37.54] SimViewer_expframe: Simulation step started
[11.0, 2:13:37.54] @@************* CPG CONTROLLER - START RESPOND OUTPUT
EVENT********************************************************************
[11.0, 2:13:37.54] CPG Controller - output event for Respond - time Elapsed: NA getTimeAdvance: 0.0 clock: 11.0
[11.0, 2:13:37.54] sendCtrl getStatus =: 6
[11.0, 2:13:37.54] Last Classical Detection Optical Power =: 0.0
[11.0, 2:13:37.54] ### CPG Controller - Sending laser fire message: CPG Controller FIRE Response Message for Case 1 Control
Message 1 to port CtrlOut2
[11.0, 2:13:37.54] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 1 Environmental Received: 0 Control Received: 1 Control
Received: 1
[11.0, 2:13:37.54] @@************* CPG CONTROLLER - END RESPOND OUTPUT
EVENT********************************************************************
[11.0, 2:13:37.54] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Internal transition
[11.0, 2:13:37.54] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Internal transition from Respond
[11.0, 2:13:37.54] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Passive for time Infinity
[11.0, 2:13:37.54] @@************* CPG CONTROLLER - START RESPOND INTERNAL
EVENT********************************************************************
[11.0, 2:13:37.69] CPG Controller - internal event for Respond - time Elapsed: NA getTimeAdvance: Infinity clock: 11.0
[11.0, 2:13:37.69] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 1 Environmental Received: 0 Control Received: 1 Control
Received: 1
[11.0, 2:13:37.69] CPG Controller - Ending Respond internal transition
```

```
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Passive for time Infinity
[11.0, 2:13:37.69] @@************* CPG CONTROLLER - END RESPOND INTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: Received messages [inCtrlIn: LaserMsg
                id: 1
                name: CPG Controller FIRE Response Message for Case 1 Control Message 1
                status: 6
                magnitude: 0.0]
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: External transition
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase UpdateLaser for time 0.0
[11.0, 2:13:37.69] @@************* LASER - START PASSIVE CTRLIN EXTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.69] Laser - external event for Passive CtrlIn with CtrlIn - time Elapsed: 11.0 getTimeAdvance: 0.0 clock: 11.0
[11.0, 2:13:37.69] Laser is set to ON
[11.0, 2:13:37.69] Laser - Preparing laser fire control message for: CPG Controller FIRE Response Message for Case 1 Control
Message 1 to port CtrlOut
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase UpdateLaser for time 0.0
[11.0, 2:13:37.69] @@************* LASER - END PASSIVE CTRLIN EXTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.69] SimViewer_expframe: Simulation step finished
[11.0, 2:13:37.288] SimViewer_expframe: Simulation step started
[11.0, 2:13:37.288] @@************* LASER - START UPDATELASER OUTPUT
EVENT**********************************************************************
[11.0, 2:13:37.288] Laser - output event for UpdateLaser - time Elapsed: NA getTimeAdvance: 0.0 clock: 11.0
[11.0, 2:13:37.288] ******************** DISPLAY PULSES IN QUEUE ********************
[11.0, 2:13:37.288] Total Number of Optical Pulses in Queue  : 0 before reflection.
[11.0, 2:13:37.288] ****************************************************************
[11.0, 2:13:37.288] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 0 Environmental Received: 0 Control Received: 1
[11.0, 2:13:37.288] @@************* LASER - END UPDATELASER OUTPUT
EVENT**********************************************************************
[11.0, 2:13:37.303] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition
[11.0, 2:13:37.319] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition from UpdateLaser
[11.0, 2:13:37.319] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase Passive for time Infinity
[11.0, 2:13:37.319] @@************* LASER - START UPDATELASER INTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.319] Laser - internal event for UpdateLaser - time Elapsed: NA getTimeAdvance: Infinity clock: 11.0
[11.0, 2:13:37.319] ******************** DISPLAY PULSES IN QUEUE ********************
[11.0, 2:13:37.319] Total Number of Optical Pulses in Queue  : 0 after reflection.
[11.0, 2:13:37.319] ****************************************************************
[11.0, 2:13:37.319] InterruptRespond = false
[11.0, 2:13:37.319] needRespond = false
[11.0, 2:13:37.319] laserpower = true
[11.0, 2:13:37.319] currentStatus = 6
[11.0, 2:13:37.319] sendCtrl status is: 6
[11.0, 2:13:37.319] Laser - Preparing optical pulse for: Laser Output Pulse 1 to port OptOut1
[11.0, 2:13:37.319] Going to Create Pulse Phase Next!!!
[11.0, 2:13:37.319] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase CreatePulse for time 5.0
[11.0, 2:13:37.319] @@************* LASER - END UPDATELASER INTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.319] SimViewer_expframe: Simulation step finished
[16.0, 2:13:37.459] SimViewer_expframe: Simulation step started
[16.0, 2:13:37.459] @@************* LASER - START CREATEPULSE OUTPUT
EVENT**********************************************************************
[16.0, 2:13:37.459] Laser - output event for CreatePulse - time Elapsed: NA getTimeAdvance: 5.0 clock: 16.0
[16.0, 2:13:37.459] ******************** DISPLAY PULSES IN QUEUE ********************
[16.0, 2:13:37.459] Total Number of Optical Pulses in Queue  : 0 BEFORE propagation.
[16.0, 2:13:37.459] ****************************************************************
[16.0, 2:13:37.459] ** OUTPUT EVENT FOR CREATEPULSE
[16.0, 2:13:37.459] ### Laser - Sending generated optical pulse: Laser Output Pulse 1 to port OptOut1
[16.0, 2:13:37.459] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 1 Environmental Received: 0 Control Received: 1
[16.0, 2:13:37.459] @@************* LASER - END CREATEPULSE OUTPUT
EVENT**********************************************************************
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition from CreatePulse
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase Passive for time Infinity
[16.0, 2:13:37.459] @@************* LASER - START CREATEPULSE INTERNAL
EVENT**********************************************************************
[16.0, 2:13:37.459] Laser - internal event for CreatePulse - time Elapsed: NA getTimeAdvance: Infinity clock: 16.0
```

```
[16.0, 2:13:37.459] ******************** DISPLAY PULSES IN QUEUE ********************
[16.0, 2:13:37.459] Total Number of Optical Pulses in Queue  : 0 BEFORE propagation.
[16.0, 2:13:37.459] ******************************************************************
[16.0, 2:13:37.459] **************** Adjust queue ****************
[16.0, 2:13:37.459] Total Number of Optical Pulses in Queue  : 0
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase Passive for time Infinity
[16.0, 2:13:37.459] @@************* LASER - END CREATEPULSE INTERNAL
EVENT**********************************************************************
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.pmfiber1: Received messages [inOptIn1: OpticalPulse
            id: 1
            name: Laser Output Pulse 1
            duration: 4.0E-10
            opticalPower: 8.709666395E-5
            brightPulseFlg: false
            amplitude: 1.94095418E-6
            centralFrequency: 1.0E15
            globalPhase: 0.0
            ellipticity: 0.0
            orientation: 0.0
            numberOfGaussians: 3
            gA0: 45.5
            gA1: 38.064
            gA2: 4.75752
            gM0: 9.54844E-11
            gM1: 1.81125E-10
            gM2: 3.52322E-10
            gSD0: 1.98151E-11
            gSD1: 5.81389E-11
            gSD2: 4.90169E-11]
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.pmfiber1: External transition
```

The following is the CPG Controller External Transition code for receiving a message when in Passive phase on port "CtrlIn$_1$" (from appendix U). If the controller receives a message while Passive on this port, it store the value of the message in variable *ctrlOutput* and checks the status type of the message by checking *ctrlMsg.status*. If this type is "init" or "get status" the output port is set to CtrlOut$_1$ or if the type is "fire laser" the port is set to CtrlOut$_2$.

```
$\delta_{ext}(phase, \sigma, store, temperature, overtemp, overpower, lastCDPower, e, ((p_i,v_i),\ldots(p_n,v_n))) =$
("respond", 0, $store, temperature, overtemp, overpower, lastCDPower$)
  if $phase$ = "passive" and $p$ = "CtrlIn$_1$"
    $ctrlOutput$ = ctrlMsg($store$)
    if $ctrlMsg.status$ = "init" or "get status"
      $outputPort$ = "CtrlOut$_1$"
    if $ctrlMsg.status$ = "fire laser"
      $outputPort$ = "CtrlOut$_2$"
```

Reviewing the CPG Controller Internal Transition code shows the controller will always output the contents of the *ctrlOutput* variable out the port contained in the *outputPort* variable. Note the (*outputPort*, *ctrlOutput*) pair is a (port,value) binary.

$\lambda(\mathit{phase}, \sigma, \mathit{store}, \mathit{temperature}, \mathit{overtemp}, \mathit{overpower}, \mathit{lastCDPower}) =$
$(\mathit{outputPort}, \mathit{ctrlOutput})$
if phase = “respond”

If the CPG Controller is Passive and receives a message on CtrlIn$_1$with status of “fire laser,” it should forward the message on port CtrlOut$_2$, as the message contents are stored in *ctrlOutput* in the external transition and output in the output phase. The laser receives this message and constructs and emits a laser pulse per its DEVS code in appendix M. Note that the test time of propagation for components is five time units, so it should take the laser five time units to produce an optical pulse. The next component in line, PMFiber$_1$, should receive the optical packet from the laser (See Figure 4).

Reviewing the MS4ME output, the controller received a LaserMsg with id:1 at time [11.0, 2:13:36.788] on port CtrlIn$_1$. The expectation is if the message is the “fire laser” type, the controller sends the fire message out port CtrlOut$_2$, which happens at time [11.0, 2:13:37.54]. The laser receives the fire message at time [11.0, 2:13:37.69] and turns the laser on to create a laser pulse. The laser takes five time units to create the optical pulse and sends it out at time [16.0, 2:13:37.459]. The next component in line, pmfiber1 receives the optical pulse.

This is the previous block of MS4ME output winnowed to show the expected behavior.

```
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Received messages [inCtrlIn1: LaserMsg
		id: 1
		name: Case 1 Control Message 1
[11.0, 2:13:36.788] CPG Controller received a laser fire message
[11.0, 2:13:37.54] ### CPG Controller - Sending laser fire message: CPG Controller FIRE Response Message for Case 1 Control
Message 1 to port CtrlOut2
```

```
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: Received messages [inCtrlIn: LaserMsg
        id: 1
        name: CPG Controller FIRE Response Message for Case 1 Control Message 1
[11.0, 2:13:37.69] Laser is set to ON
[11.0, 2:13:37.319] Laser - Preparing optical pulse for: Laser Output Pulse 1 to port OptOut1
[11.0, 2:13:37.319] Going to Create Pulse Phase Next!!!
[16.0, 2:13:37.459] ### Laser - Sending generated optical pulse: Laser Output Pulse 1 to port OptOut1
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.pmfiber1: Received messages [inOptIn1: OpticalPulse
        id: 1
        name: Laser Output Pulse 1
```

Each test case is reviewed in the same manner to ensure the captured output matches the expected behavior for each submodule controller, test case, and component behavior. Component behavior is referenced from previously captured output during the component testing.

8.3.3 *Closure Under Coupling*

One of the significant prosperities of DEVS is closure under coupling. Simply stated, if any result from coupling components specified by the formalism can itself be specified by the formalism, then it has the closure under coupling property (Barros, 1997; B. P. Zeigler, 1984). Chow showed this property existed in the Parallel-DEVS formalism he specified in 1996 (Chow, 1996).

The coupled submodules discussed earlier in this chapter exhibited this property during testing in the MS4ME simulator. DEVS requires explicit definition of each message type or input, so each coupled model only accepts a finite set of inputs and the outputs are definable under Parallel-DEVS, thereby demonstrating closure under coupling.

8.4 *Summary*

This chapter presented the results and analysis from creation and testing methodology the atomic and coupled DEVS modules for the *qkdX* architecture. This research effort created, modeled and tested seventeen optical components and seven coupled submodules by the end of the research period. The research produced a combined model output exceeding 7,300 pages and almost 200,000 lines of MS4ME code, with almost five times as much code devoted to testing than for the models themselves.

As there was no automated testing available, each line of output had to be checked with the operational simulator graphics, compared to the lines of model code and the DEVS pseudocode, based on the mathematical model supplied by the optical SME. Finally, after successful testing, the results were passed to the optical SME for final checking to ensure the proper behavior and timing was captured in the DEVS pseudocode and the MS4ME models. This was a cooperative process throughout, with the optical SME and experts from the AFIT QKD team constantly reviewing the progress to provide feedback on the modeling effort.

As the models developed, they evolved closer to the expected system behavior, until each was deemed acceptable by the SMEs and research team. Using the definitions of validity discussed earlier, this effort provided models that captured the required behavior and met the required accuracy, and so are considered “valid” with the understanding this validity only applies to the models built for the specific experimental frame. These “acceptable” and “valid” models increase the academic rigor of the simulation framework by providing a coherent conceptual modeling methodology using a proven modeling formalism that demonstrates the required and expected behavior.

Using DEVS allowed the team to refine the simulation framework, correcting several errors and aiding the research team in recognizing missing behaviors within the simulation (see chapter 7). DEVS increased the validity of QKD simulation framework optical pathway by showing DEVS can be used to increase validity by creating optical component models fit for the purposes of the simulation and acceptable to the community of developers and users.

9. Conclusions

9.1 *Chapter Overview*

The primary objective of this research was to explore the use of the DEVS to create conceptual models of optical components for use in the *qkdX* framework. The goal was to use the simulation study process, conceptual model and validity theory, and DEVS to contribute to the QKD sponsored research project. This chapter presents the conclusions drawn from the research and recommendations for future work.

9.2 *Conclusions of Research*

The research into DEVS and conceptual modeling highlighted these main points:

9.2.1 *DEVS Forces 'real-time' DES*

The accepted paradigm in DES simulations is to schedule events in the future. As the DES moves from one state to the next, this list is checked for the next occurring event and the time in the simulator jumps to the event execution time. This makes DES efficient, as the simulation 'skips' over time units where there are no events. The problem occurs when events cause a change that affects future events. The simulator must be able to correct or undo any future event, even if it means clearing the entire future event list and recalculating every event. In this situation, the user is relying on the simulator to properly correct the future event list, expecting no mistakes or unforeseen simulation behavior from these changes.

DEVS differs from this future event list by not having 'future time.' DEVS has two types of time: 1) time the state will stay static; 2) time elapsed since the last state transition. The modeler cannot schedule events in the future and must specify the state transitions from one state to the next. This precludes the simulation from entering an unexpected state and prevents emergent behavior. The system can never enter a state that is unrecoverable (unless that was the intent) and so prevents the Once Passive, Never Active (OPNA) condition that is seen in complex DES simulations.

9.2.2 *Discovery of Non-composability in qkdX*

One of the sponsor requirements for the QKD simulation framework is that of composability. The simulation must be able to change components and modules without extensive code change. The design concept was for a “plug and play” simulation where users can easily change system designs. This requirement is reflected in the user and developer requirements discussed earlier in this dissertation. Ensuring composability is a major undertaking for the research project.

During the creation of the conceptual models for each of the optical components and coupled submodules led to discovery of problems within the original version of the *qkdX* simulation framework. This version was built straight from the SME mathematical models as a proof-of-concept demonstration without using formalized conceptual models, leaping from the SME modeling level to the simulation coding level of *Figure 1* and using a “mental model” as a bridge between the math model and the final simulation. Time and resource considerations, along with the intent to only produce a demonstration simulation, where among the reasons for choosing this development course. See Figure 5.

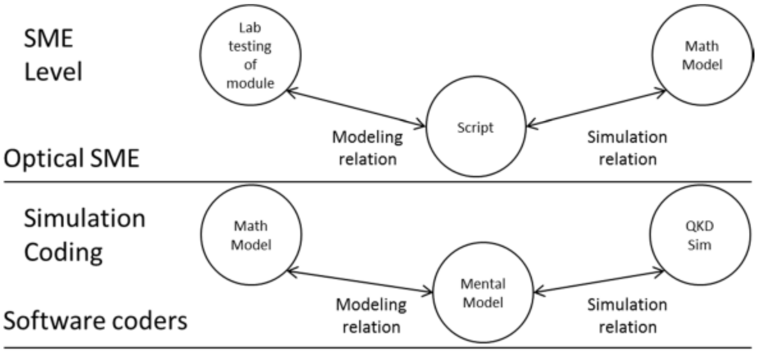


Figure 5. Abbreviated Levels of Modeling for *qkdX*.

Several of these problems would have prevented the optical models from executing the same, regardless of how they were connected, while others were incorrect expressions of component behavior. Using DEVS forces the modeler to carefully consider how each model behaves under all conditions, and guarantees the models exhibit *closure under coupling* (see chapter 7). The careful consideration of behavior required by DEVS decreases the likelihood of errors or missing behaviors during the conceptual modeling phase.

This is not to mean that having complete DEVS models ensures the *simulation* exhibits composability. While the modeling relation and validity measures the closeness of the conceptual model to the source system, the simulation relation and verification is a measure of how well the simulation executes the behavior of the conceptual models (See Figure 6, modified from (B. P. Zeigler, Praehofer, & Kim, 2000)).

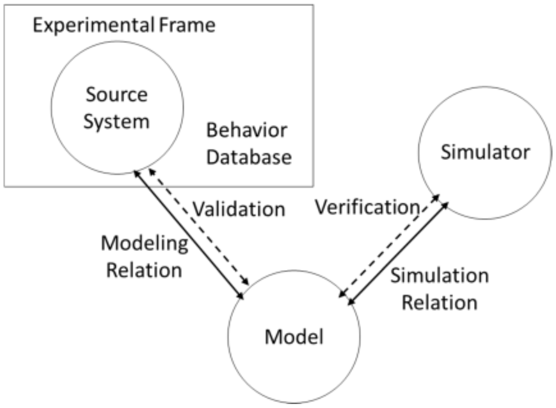


Figure 6. Modified from “Basic entities in M&S and their relationships”.

The skills, talents, and experience of the modelers responsible for turning the conceptual modeling into an executable simulation, along with verification testing, determine if the final simulation exhibits composability. In this case, composable DEVS models do not guarantee *qkdX* is itself composable. The closer the simulation executes the required behavior, the greater the chance of composability and by using DEVS, the better chance of capturing all required behavior.

9.2.3 *Well-defined Behavior*

A concern for complex DES simulation is ill-defined or emergent behavior that leads to the simulation entering a state or condition it cannot leave (known as OPNA). Simple simulations can be checked by evaluating every possible state, but with a large simulation, the state-space becomes unmanageable and infeasible to check. Project time and funds limit testing for simulations, even with automated support.

In contrast, using DEVS for modeling prevents these types of problem behaviors. DEVS requires the modeler to identify the lowest level of objects in the simulation, one that cannot be further decomposed (*atomic*). These atomic models can be modeled regardless of their immediate context (i.e. the component is isolated from any outside influences) and their total behavior specified. A correctly specified model operates properly regardless of its use and the environment. By fully specifying the atomic behavior, the modeler is assured the components work properly when connected, in any possible combination (*closure under coupling*). This allows for hierarchal construction of models and ensures modularity of the simulation.

9.2.4 *Tool-Independent Form*

A major consideration in simulation creation is selecting the software environment. This choice can have long-term effects on the simulation, as a poor or hasty choice can lead to a failed project. This selection process was a secondary research question discussed in chapter 2 and discussed in chapter 5. Some simulation environments have their own programming language while others use standard programming languages (i.e. OMNeT++ uses the C++ language). Each language may have idiosyncrasies that prevent easy translation from one simulation environment to another (Miller, 2013a). For an example, the Java language (used in MS4ME) has routines to free memory storage (garbage collection) but C++ does not; the Java runtime system has ways of knowing the size of an array, but the C++ runtime does not (Miller, 2013a; Miller, 2013b; Miller, 2013c). Understanding these examples requires some experience in programming, but do highlight there are language-specific idiosyncrasies that prevent simple conversations.

DEVS forces the modeler to consider carefully all facets of desired behavior within the model, including all inputs, outputs, and timing segments. The resulting models are based solely on the language rules inherent in DEVS, and not on the selected simulation environment. This allows for creation of models usable for any simulation environment. The modeler needs only to express the DEVS behavior in the simulator of choice.

9.2.5 *Canonical Behavior*

DEVS forces the modeler to consider carefully all aspects of the conceptual model, starting with the inputs, outputs and behaviors of interest necessary for study. This *experimental frame* greatly reduces or eliminates emergent or unexpected behavior. By considering 'real-time' transitions, well-defined behavior and tool-independence, the modeler creates a deep understating of the system usable in any modeling simulation. By understanding the *canonical behavior* of the model, the modeler can implement changes to decrease the differences between the model and the referent system, thereby increasing the validity of the model for the chosen experimental frame.

9.2.6 *Difficulties of Modeling Using DEVS*

Much of the written material available for DEVS in textbooks and papers provides only simplistic examples. Even the more complex examples available through RTSync provided little guidance to model the optical components. This complexity led this researcher to contact the co-director of Arizona Center for Integrative Modeling and Simulation (Arizona State University, 2014), the research center where DEVS was created, for advice on using DEVS for our unique, innovative models. The co-director provided suggestions on how to use DEVS to model the timing issues at the picosecond scale and capturing wave and particle behavior.

It can be difficult to verify the DEVS pseudocode to the source system behavior, especially true when modeling predicted or notional systems. Since the demonstration QKD system is built from real optical components but in a notional architecture, there was the difficulty of verifying the component behavior to real components. The solution used in this research to this problem greatly increased the DEVS work by necessitating programming the pseudocode twice, once into MS4ME, and then into the selected qkdX simulator.

DES models are increasing in complexity and being used in new information-centered fields. DEVS was not created for these types of problems and has difficulty expressing processes and information flows within the formalism. Heretofore, the solution has been to create subsets of DEVS to handle specific problems. This is leading to a fragmentation of the formalism and makes it hard to determine which form of DEVS is appropriate for the modeling problem. This research used the Parallel-DEVS formalism for modeling the optical path, but this DEVS may not be applicable to modeling the processes found in QKD systems.

DEVS is a set of language rules to describe formally a problem. There is no visual component to DEVS unless the modeler uses a DEVS-compliant simulation program. While useful for being tool-independent, this requires the modeler to use that particular simulator’s functions, which may not necessarily conform completely to DEVS, as seen with issues in EVOA timing and the MS4ME simulator (see chapter 7).

9.3 *Recommendations for Future Research*

This research investigated using the DEVS to create conceptual models for a prototypical QKD architecture. The scope was intentionally narrowed to the optical path of this particular architecture, rather than to try to model every identified optical component. During the research period, the researcher identified additional areas for research using the DEVS formalism.

The AFIT QKD team continues to build on the *qkdX* framework, adding new capabilities to model other types of QKD systems. This research used weak-coherent polarization-based system architecture, the same kind as the first QKD system. While useful as a reference baseline, this type is rarely found outside of academic research systems as most commercial QKD systems use a phase-based architecture. The AFIT team is adding the components and functions necessary to model these systems, as each piece of new hardware needs a corresponding conceptual model, so there is a continuing need to create conceptual models.

An intriguing area is using DEVS to express the functions and processes inherent in each type of QKD system. This research focused on the optical path hardware in the quantum modules, but QKD systems have many processes in each protocol. DEVS works well with state machine simulations, but it is unknown if it could be used to create conceptual models for complex processes. During the literature review for this research, no examples were found using DEVS to model processes, but as there is continuing research into DEVS, there may a process-compatible version in the future.

This research used the MS4ME DEVS simulator as a test platform for the DEVS pseudocode. Using traces of the output allowed for checking of component behavior.

Unfortunately, there was no automated tool for checking the output, so the researcher manually reviewed over 7000 pages of output. This took considerable time and effort that slowed the research process by taking time away from other research activities. An automated tool for checking this output would have considerable impact on the time necessary to perform traces on component output.

One of the areas included for research during the prospectus was to compare output from the MS4ME simulator to output from the *qkdX* simulation. This would be exploration of the *modeling relation* (see Figure 2 in chapter 7) between the conceptual model and the simulator. Work in this area ceased when the researcher and the software SMEs realized: 1) exploring this relation was outside the scope of research into model validation; 2) considerable time and effort was necessary to enable this functionality, as it required extensive changes to both simulators. An automated comparison tool would save considerable effort on the part of researchers and allow for a standardized process to test the modeling relation between the conceptual model and the QKD simulation.

As mentioned in chapter 2, the AFIT QKD team received a QKD system late into this research period. A future research project could be to model the real system using DEVS and MS4ME and compare the output between the real system and the MS4ME models. This comparison between the conceptual models and a real system would be an additional V&V technique to increase the validity of the DEVS models.

The work presented in this document is a first step in expressing optical components using the DEVS formalism and increasing the validity of the *qkdX* simulation framework. The results demonstrated that DEVS can create conceptual models of continuous-time optical components in a DES environment. This work showed that DEVS can be used to increase the validity of the *qkdX* simulation thus improving the quality of the simulation and possibly increasing its d value to the project sponsor and future end-users.

10. References

Arizona State University. (2014). Arizona center for integrative modeling and simulation. Retrieved, 2014, Retrieved from http://acims.asu.edu/

Austrian Institute of Technology. (2014). Qkd software. Retrieved, 2014, Retrieved from https://sqt.ait.ac.at/software/ projects/qkd-software

Balci, O. (1995). Principles and techniques of simulation validation, verification, and testing. Paper presented at the *Proceedings of the1995 Winter Simulation Conference*, 147-154. Retrieved from http://www.informs-sim.org/wsc95papers/1995_0021.pdf

Balci, O. (1997). Verification validation and accreditation of simulation models. Paper presented at the *Proceedings of the 29th Winter Simulation conference*, 135-141. Retrieved from http://www.informs-sim.org/wsc97papers/0135.PDF

Banks, J. (Ed.). (1998). *Handbook of simulation*. New York: John Wiley & Sons.

Banks, J., Carson, J. S., Nelson, B. L., & Nicol, D. M. (2010). *Discrete event system simulation* (5th ed.). Englewood Cliffs, NJ: Prentice Hall.

Banks, J., & Chwif, L. (2010). Warnings about simulation. *Journal of Simulation, 5*(4), 279-291. Retrieved from http://www.simulate.com.br/warnings.pdf

Banks, J., & Gibson, R. R. (1997). Don't simulate when…. *IIE Solutions, 9*, 30-32. Retrieved from http://www.blueminegroup.com/aai/pdf/10_Rules_Determining.pdf

Banks, J., & Gibson, R. R. (2001). Simulating in the real world. *IIE Solutions, 33*(4), 38-40. Retrieved from http://www.blueminegroup.com/aai/pdf/Simulating_RealWorld.pdf

Barker, E. B., Barker, W. C., & Lee, A. (2005). *NIST special publication 800-21 guideline for implementing cryptography in the federal government* NIST. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-21-1/sp800-21-1_Dec2005.pdf

Barros, F. J. (1997). Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS), 7*(4), 501-515. Retrieved from http://estudogeral.sib.uc.pt/jspui/bitstream/10316/10705/1/Modeling%20formalisms%20for%20dynamic%20structure%20systems.pdf

Bellovin, S. M. (2011). Frank miller: Inventor of the one-time pad. *Cryptologia, 35*(3), 203-222. Retrieved from http://academiccommons.columbia.edu/download/fedora_content/download/ac:135404/CONTENT/cucs-009-11.pdf

Chow, A. C. (1996). Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation, 13*(2), 55-68. Retrieved from http://www.bgc-jena.mpg.de/~twutz/devsbridge/pub/chow96_parallelDEVS.pdf

Elliott, R., Edmondson, D., Scrudder, R., Igarza, J., & Smith, N. (2009). *Manager's guide to the high level architecture for modeling and simulation (HLA).* Unpublished manuscript. Retrieved 3/25/2013, Retrieved from http://www.msco.mil/documents/_2_ITEC09%20-%20HLA%20Tutorial.pdf

Geoffrion, A. M. (1989). Integrated modeling systems. *Computer Science in Economics and Management, 2*(1), 3-15. Retrieved from http://www.dtic.mil/dtic/tr/fulltext/u2/a215219.pdf

Gogg, T. J., & Mott, J. R. (1998). Introduction to simulation. Paper presented at the *Proceedings of the 1993 Winter Simulation Conference,* 9. Retrieved from http://www.informs-sim.org/wsc93papers/1993_0002.pdf

Grimaila, M. R., Morris, J. D., & Hodson, D. (2012). Quantum key distribution, a revolutionary security technology. *The ISSA Journal, 27*, 20-27.

Gunay, H. B., O'Brien, L., Goldstein, R., Breslav, S., & Khan, A. (2013). Development of discrete event system specification (DEVS) building performance models for building energy design. Paper presented at the *Proceedings of the Symposium on Simulation for Architecture & Urban Design,* San Diego, CA. 22-30. Retrieved from http://www.autodeskresearch.com/pdf/46_Final_Paper.pdf

Jain, R. (1991). *The art of computer systems performance analysis*. New York: John Wiley & Sons.

Law, A. M., Kelton, W. D., & Kelton, W. D. (1991). *Simulation modeling and analysis* (2nd ed.). New York: McGraw-Hill.

Miller, J. R. (2013a). Moving from java to c++. Retrieved, 2014, Retrieved from http://people.eecs.ku.edu/~miller/Courses/JavaToC++/JavaToC++.html

Miller, J. R. (2013b). Moving from java to c++: Arrays. Retrieved, 2014, Retrieved from http://people.eecs.ku.edu/~miller/Courses/JavaToC++/Arrays.html

Miller, J. R. (2013c). Moving from java to c++: Memory management. Retrieved, 2014, Retrieved from http://people.eecs.ku.edu/~miller/Courses/JavaToC++/MemoryManagement.html

Mittal, S., Risco, J. L., & Zeigler, B. P. (2007). DEVS-based simulation web services for net-centric T&E. Paper presented at the *Proceedings of the 2007 Summer Computer Simulation Conference*, San Diego, CA. 357-366. Retrieved from http://cell-devs.sce.carleton.ca/citations/SC208_Mittal.pdf

Morris, J. D., Grimaila, M. R., Hodson, D. D., Jacques, D., & Baumgartner, G. (2013). A survey of quantum key distribution (qkd) technologies. In B. Akhgar, & H. R. Arabnia (Eds.), *Emerging trends in ICT security* (1st ed., pp. 141-152). Waltham, MA: Elsevier.

Morris, J. D., Hodson, D. D., Grimaila, M. R., Jacques, D. R., & Baumgartner, G. (2014). Towards the modeling and simulation of quantum key distribution systems. *International Journal of Emerging Technology and Advanced Engineering, 4*(2)

Morse, K. L., Coolahan, J. E., Lutz, B., Horner, N., Vick, S., & Syring, R. (2010). *Best practices for the development of models and simulations.* (Final Report No. 2010-307). Laurel, MD: John Hopkins University. Retrieved from http://www.msco.mil/documents/10-S-2_26_952%20-%20SIW10F%20-%20MS%20Development%20Best%20Practices%20Final%20Report%20-%20Diem%20-%2020100812%20-%20Dist%20A%20(3).pdf

MS4 Systems. (2014). MS4ME. Retrieved, 2014, Retrieved from http://www.ms4systems.com/pages/ms4me.php

Naylor, T. H., & Finger, J. M. (1967). Verification of computer simulation models. *Management Science, 14*(2), B-92-B-101. Retrieved from http://mansci.journal.informs.org/content/14/2/B-92.full.pdf

Ntaimo, L., Zeigler, B. P., Vasconcelos, M. J., & Khargharia, B. (2004). Forest fire spread and suppression in DEVS. *Simulation, 80*(10), 479-500. Retrieved from http://ise.tamu.edu/people/faculty/Ntaimo/personal_web/Papers/NtaimoSIM012004.pdf

Sargent, R. G. (2005). Verification and validation of simulation models. Paper presented at the *Proceedings of the 37th Conference on Winter Simulation*, 130-143. Retrieved from http://student.telum.ru/images/6/66/Sargent_VV_2010.pdf

Scarani, V., Bechmann-Pasquinucci, H., Cerf, N. J., Dušek, M., Lütkenhaus, N., & Peev, M. (2009). The security of practical quantum key distribution. *Reviews of Modern Physics, 81*(3), 1301. Retrieved from http://arxiv.org/pdf/0802.4155

Schneier, B. (1995). In Sutherland P. (Ed.), *Applied cryptography: Protocols, algorithms, and source code in C* (2nd ed.). New York: John Wiley & Sons, Inc.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, *27*, 379-423. Retrieved from http://www.inf.ed.ac.uk/teaching/courses/com/handouts/extra/shannon-1948.pdf

Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell System Technical Journal*, *28*(4), 656-715. Retrieved from http://dm.ing.unibs.it/giuzzi/corsi/Support/papers-cryptography/Communication_Theory_of_Secrecy_Systems.pdf

Shannon, R. E. (1998). Introduction to the art and science of simulation. Paper presented at the *Proceedings of the 1998 Winter Simulation Conference,* Washington DC. , *1* 7-14. Retrieved from http://www.reocities.com/ramonroque/Articulo_Introduction_to_the_art_and_science_of_simulation.pdf

Singh, S. (1999). *The code book: The secret history of codes and code-breaking* (17th ed.). London: Fourth Estate.

Wainer, G. (2006). Applying cell-DEVS methodology for modeling the environment. *Simulation, 82*(10), 635-660. Retrieved from http://cell-devs.sce.carleton.ca/publications/2006/Wai06/Environment635.pdf

Wainer, G. A., & Giambiasi, N. (2001). Application of the cell-DEVS paradigm for cell spaces modelling and simulation. *Simulation*, *76*(1), 22-39. Retrieved from http://cell-devs.sce.carleton.ca/papers/Simulation01.pdf

Wolfram. (2014). Wolfram mathematica. Retrieved, 2014, Retrieved from http://www.wolfram.com/mathematica/

Zeigler, B. P. (1984). *Multifaceted modelling and discrete event simulation*. London, UK: Academic Press.

Zeigler, B. P., Kim, D., & Buckley, S. J. (1999). Distributed supply chain simulation in a DEVS/CORBA execution environment. Paper presented at the *Proceedings of the 31st Winter Simulation Conference,* 1333-1340. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.9681&rep=rep1&type=pdf

Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems* (2nd ed.). San Diego: Academic Press.

Zeigler, B. P., Sarjoughian, H. S., & Au, V. (1997). Object-oriented DEVS: Object behavior specification . Paper presented at the *Proceedings of Enabling Technology for Simulation Science,* Orlando, Fl. , *3083* 100-111.

Zeigler, B. P., Ball, G., Cho, H., Lee, J., & Sarjoughian, H. (1999). Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions. Paper presented at the *Simulation Interoperation Workshop,* (99S-SIW) 065-073. Retrieved from http://acims.asu.edu/wp-content/uploads/2012/02/SIWDEVSImplemHLARTI.pdf

Zeigler, B. P., Song, H. S., Kim, T. G. & Praehofer, H. (1995). DEVS framework for modelling, simulation, analysis, and design of hybrid systems.

Appendix A - QKD Prototypical Architecture

This appendix contains the systems engineering material created during the selection of the prototypical QKD system architecture. Much of the material derives from the Department of Defense Architectural Framework (DODAF), currently in version 2.02 (Department of Defense, 2010). The DODAF is a comprehensive framework and conceptual model that enables architectural development and allows for key decisions through organized information sharing. The framework is a group of interrelated models and views of the project that allow for accountability and traceability throughout the design process. The following table lists the DODAF views presented in this research.

Table 1. *List of DODAF Views*.

DODAF Views	**Equivalent Work**
AV-1 Overview and Summary Information	Completed, not included
AV-2 Integrated Dictionary	Continuing updates, not included
OV-1 High-Level Operational Concept Graphic	QKD Context Diagram
OV-2 Operational Resource Flow Description	High level Resource Flows Diagram
OV-5a Operational Activity Decomposition Tree	Activity Decomposition Diagram
OV-5b Operational Activity Model	Activity Models
SV-1 Systems Interface Description	System Interface Graphic
SV-2 Systems Resource Flow Description	System Resource Flows Diagrams
SV-5a Operational Activity to Systems Function Traceability Matrix	Activity to Systems Function Matrix
SV-5b Operational Activity to Systems Traceability Matrix	Subsystem to Activity Matrix
SV-10a Systems Rules Model	Use cases for components and modules in each appendix
SV-10b Systems State Transition Description	Phase transition diagrams in each appendix
SV-10c Systems Event-Trace Description	Event-trace tables and diagrams in each appendix

A.1 Systems Engineering Support

The information presented here is support for the design decisions during the building of the prototypical architecture, but was not included for discussion in the articles presented in

Chapters 6 and 7, as neither of the articles focused on the systems engineering aspects of the architecture. The end of this chapter presents the tools used during the research.

A.1.1 QKD Context

This graphic shows the high-level concepts for the QKD system. Here Alice uses quantum exchange to pass information to Bob. Through the phases of sifting, error correction and privacy amplification, a final key is generated and distributed to their respective encryptors

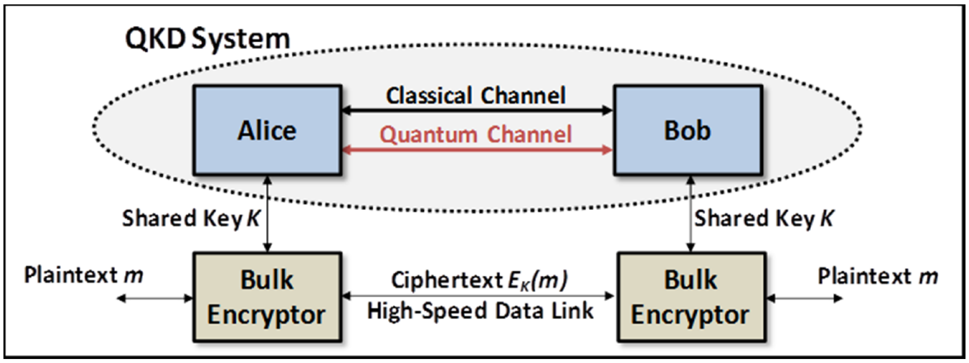
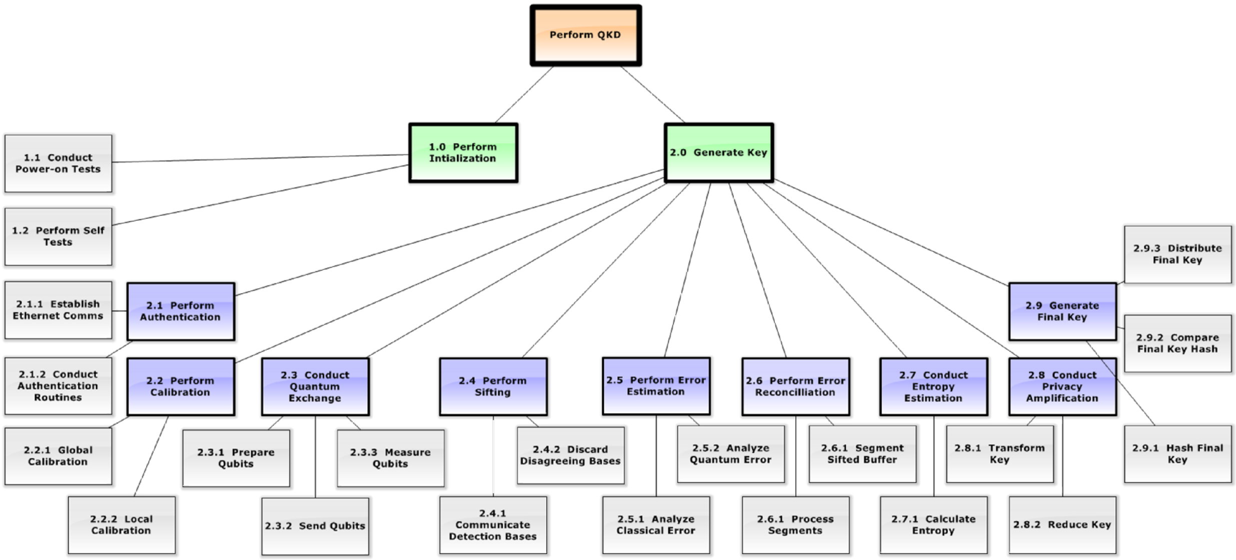


Figure 1. QKD Context Diagram.

A.1.2 Activity Decomposition

This graphic shows the decomposition of the BB84 QKD system into levels of operational activities. Each level is color-coded for clarity. The QKD activity is decomposed into two activities, one that handles the initialization of the system and one that handles key generation and distribution.



Perform QKD
1.0 Perform Intialization
2.0 Generate Key
1.1 Conduct Power-on Tests
1.2 Perform Self Tests
2.1.1 Establish Ethernet Comms
2.1 Perform Authentication
2.1.2 Conduct Authentication Routines
2.2 Perform Calibration
2.3 Conduct Quantum Exchange
2.4 Perform Sifting
2.5 Perform Error Estimation
2.6 Perform Error Reconcilliation
2.7 Conduct Entropy Estimation
2.8 Conduct Privacy Amplification
2.9 Generate Final Key
2.9.3 Distribute Final Key
2.9.2 Compare Final Key Hash
2.9.1 Hash Final Key
2.2.1 Global Calibration
2.2.2 Local Calibration
2.3.1 Prepare Qubits
2.3.2 Send Qubits
2.3.3 Measure Qubits
2.4.1 Communicate Detection Bases
2.4.2 Discard Disagreeing Bases
2.5.1 Analyze Classical Error
2.5.2 Analyze Quantum Error
2.6.1 Process Segments
2.6.1 Segment Sifted Buffer
2.7.1 Calculate Entropy
2.8.1 Transform Key
2.8.2 Reduce Key

Figure 2. QKD Activity Decomposition

A.1.3 Activity Model – Generate Key

This graphic represents the resource flows within the Generate Key activity seen in the activity decomposition diagram.

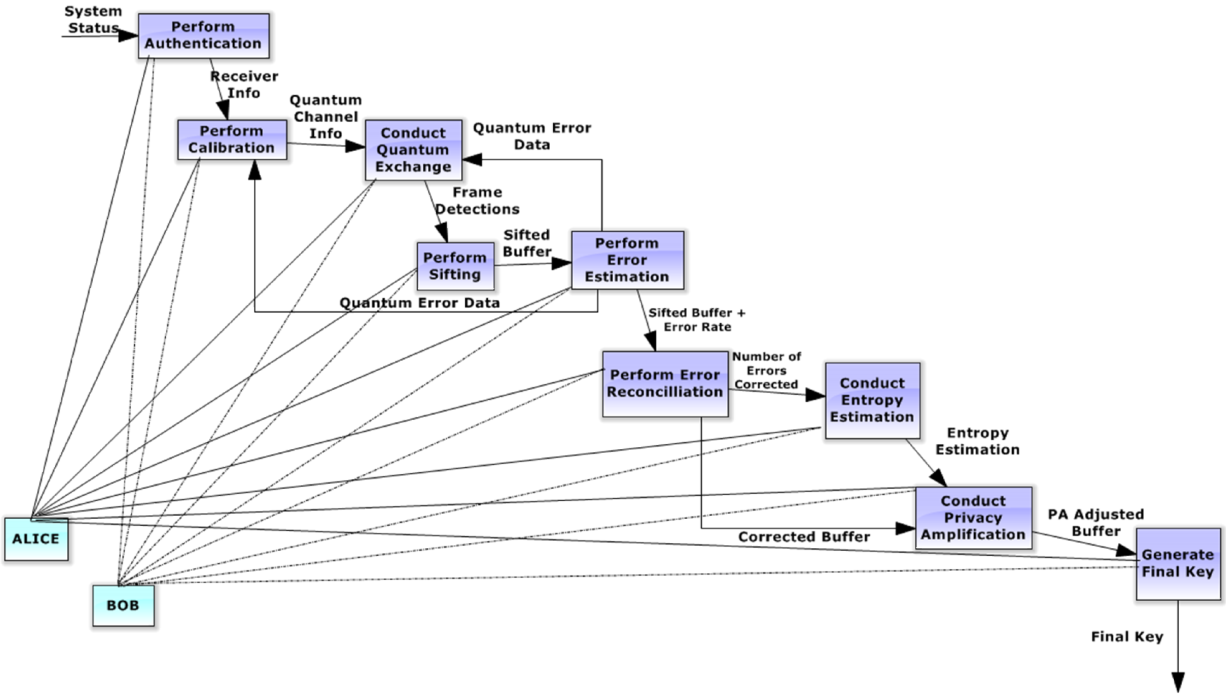


Figure 3. Generate Key Activity Model.

A.1.4 Activity Model Decomposition– Conduct Quantum Exchange

This graphic represents the decomposition of the Conduct Quantum Exchange activity seen in the activity decomposition diagram.

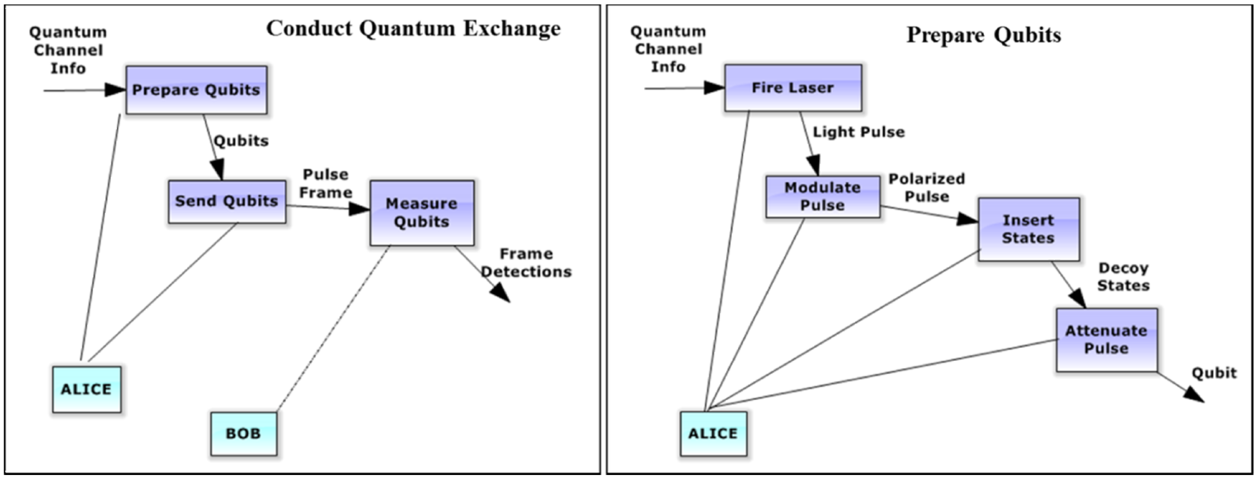


Figure 4. Decomposition of Conduct Quantum Exchange Activities part 1.

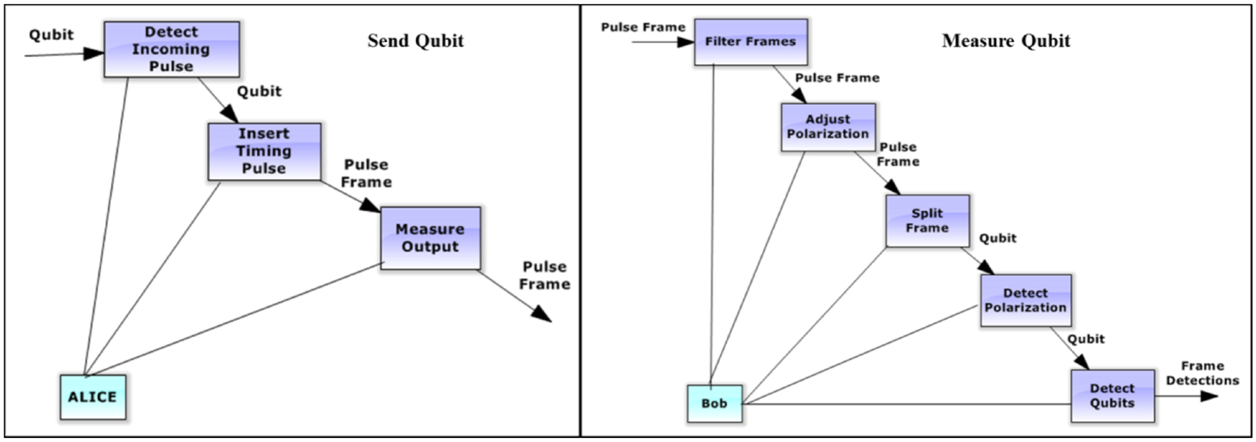


Figure 5. Decomposition of Conduct Quantum Exchange Activites part 2.

A.1.5 High Level Resource Flows

This graphic shows the resources flowing between Alice and Bob within a QKD system and flowing outside to their respective encryptors. Note that the encryptors are outside the considered system.

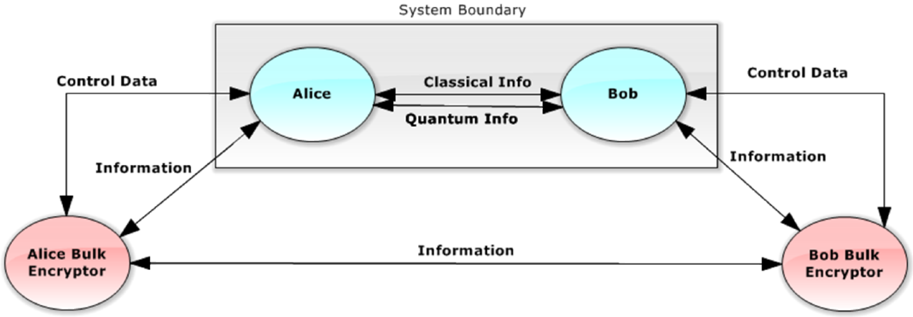


Figure 6. QKD Resource Flows.

*A.1.6 Systems Interfac*e *Graphic*

This graphic show the subsystems located within Alice and Bob and the communication channels between each subsystem. This research concentrates on the Alice Quantum Module.

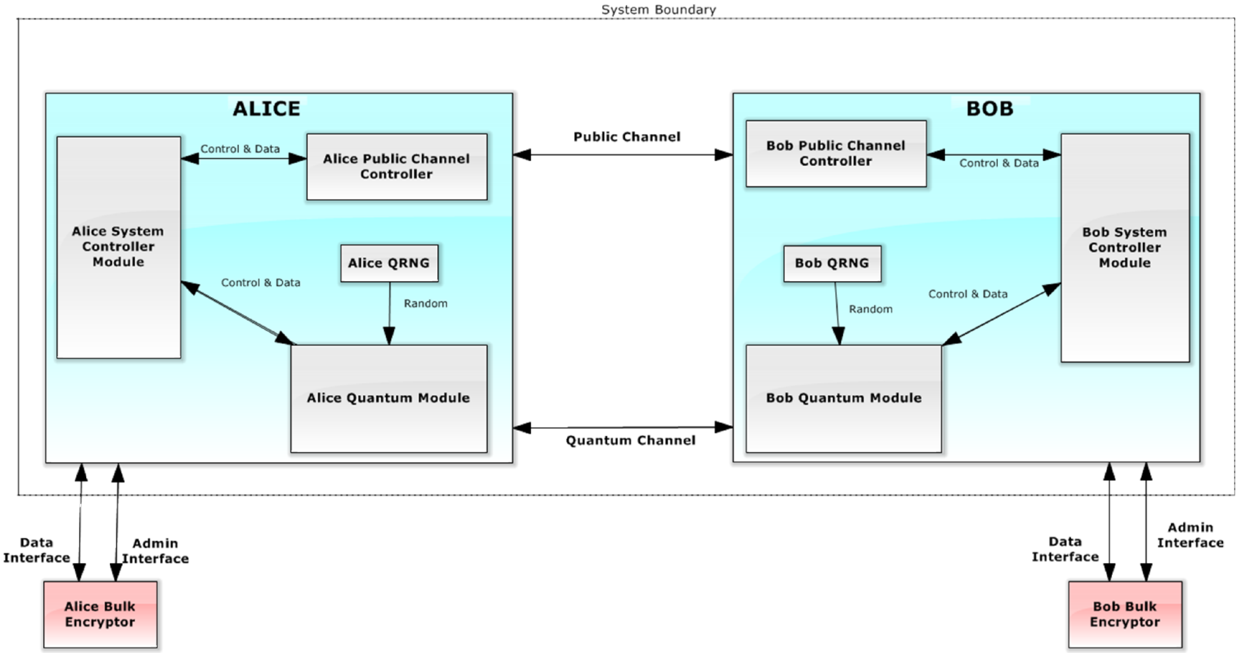


Figure 7. Alice Systems Interfaces.

A.1.7 Systems Resource Flows – Alice Quantum Module

This graphic shows the sub modules within the Alice Quantum Module and the data resource flow between each. Note the Environmental message channels (ENV) shown here enable the passing of environmental data to each module from the simulation engine and enable the modules to react to environment changes. These channels are present at every level and are not shown on the following diagrams for clarity.

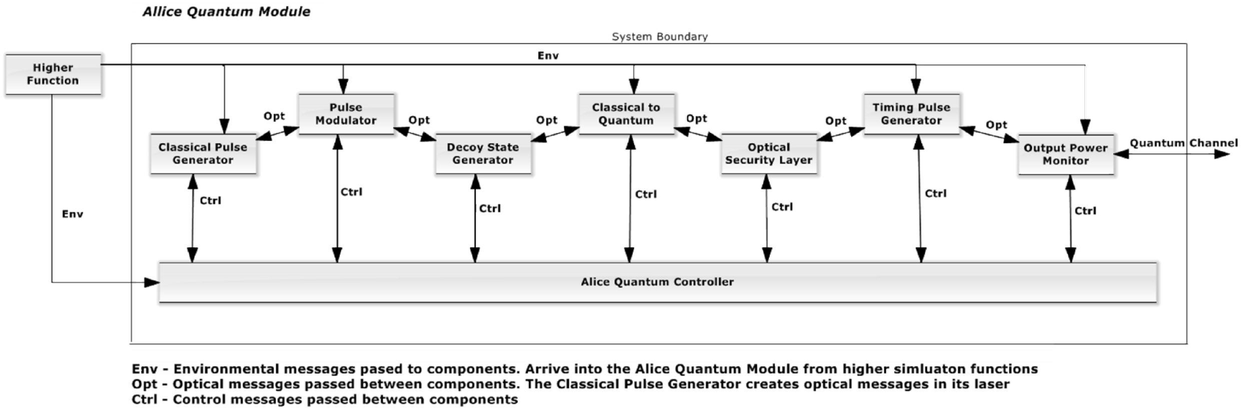

Allice Quantum Module
System Boundary
Env
Higher Function
Pulse Modulator
Classical to Quantum
Timing Pulse Generator
Opt
Opt
Opt
Opt
Opt
Opt
Classical Pulse Generator
Decoy State Generator
Optical Security Layer
Output Power Monitor
Quantum Channel
Ctrl
Ctrl
Ctrl
Ctrl
Ctrl
Ctrl
Ctrl
Env
Alice Quantum Controller
Env - Environmental messages pased to components. Arrive into the Alice Quantum Module from higher simluaton functions
Opt - Optical messages passed between components. The Classical Pulse Generator creates optical messages in its laser
Ctrl - Control messages passed between components

Figure 8. Alice Quantum Module Resource Flows.

The following graphics show the decomposition of the Alice Quantum Module subsystems and identify the individual optical components within subsystem. The graphics show the types of data flowing between each component. Chapters 6 & 7 have detailed discussions on the creation and modeling of each sub module and component. Note each component has an ENV channel as noted earlier.

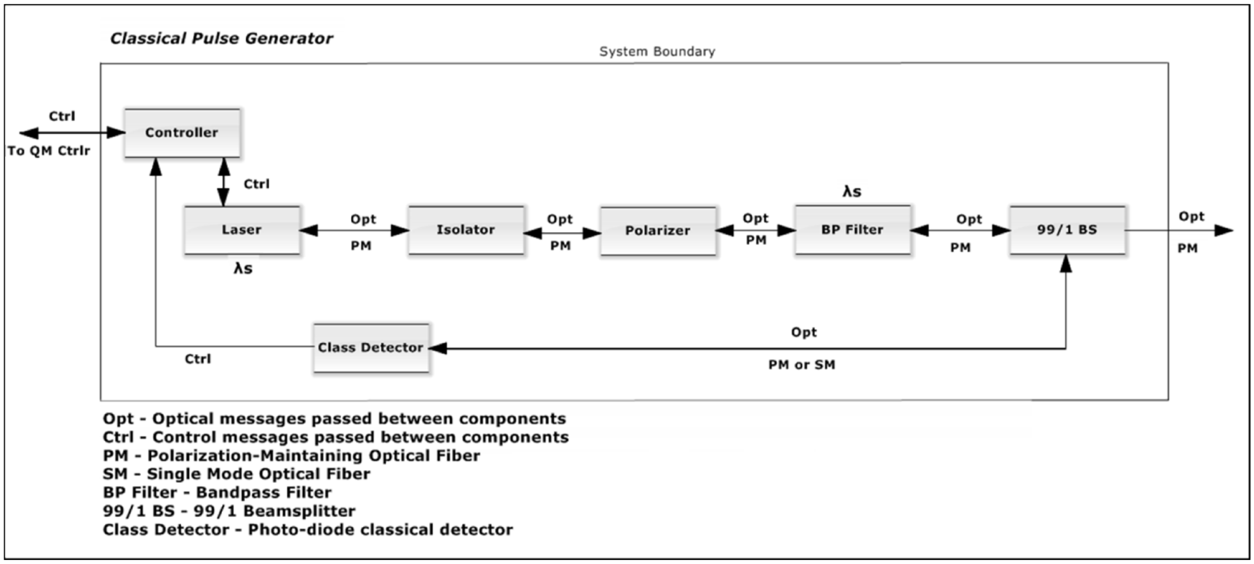

Classical Pulse Generator
System Boundary
Ctrl
To QM Ctrlr
Controller
Ctrl
Laser
λs
Opt
PM
Isolator
Opt
PM
Polarizer
Opt
PM
λs
BP Filter
Opt
PM
99/1 BS
Opt
PM
Class Detector
Ctrl
Opt
PM or SM
Opt - Optical messages passed between components
Ctrl - Control messages passed between components
PM - Polarization-Maintaining Optical Fiber
SM - Single Mode Optical Fiber
BP Filter - Bandpass Filter
99/1 BS - 99/1 Beamsplitter
Class Detector - Photo-diode classical detector

Figure 9. Classical Pulse Generator Module.

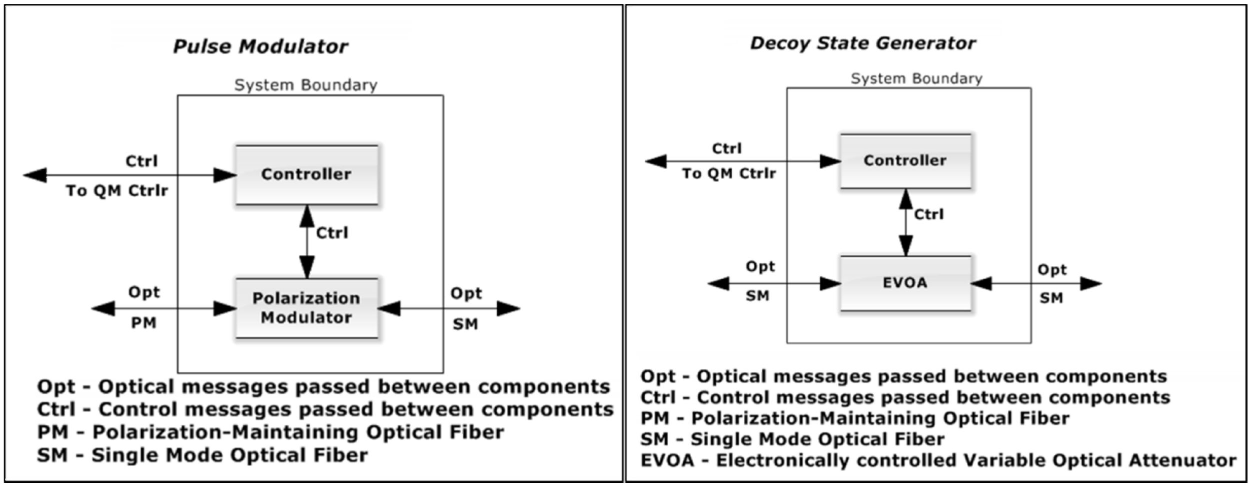

Pulse Modulator
System Boundary
Ctrl
To QM Ctrlr
Controller
Ctrl
Opt
PM
Polarization Modulator
Opt
SM
Opt - Optical messages passed between components
Ctrl - Control messages passed between components
PM - Polarization-Maintaining Optical Fiber
SM - Single Mode Optical Fiber
Decoy State Generator
System Boundary
Ctrl
To QM Ctrlr
Controller
Ctrl
Opt
SM
EVOA
Opt
SM
Opt - Optical messages passed between components
Ctrl - Control messages passed between components
PM - Polarization-Maintaining Optical Fiber
SM - Single Mode Optical Fiber
EVOA - Electronically controlled Variable Optical Attenuator

Figure 10. Pulse Modulator and Decoy State Generator Modules.

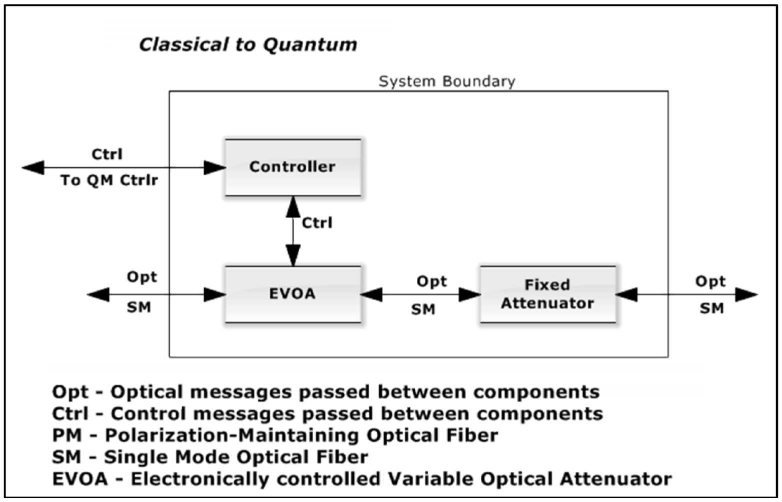

Classical to Quantum
System Boundary
Ctrl
To QM Ctrlr
Controller
Ctrl
Opt
SM
EVOA
Opt
SM
Fixed Attenuator
Opt
SM
Opt - Optical messages passed between components
Ctrl - Control messages passed between components
PM - Polarization-Maintaining Optical Fiber
SM - Single Mode Optical Fiber
EVOA - Electronically controlled Variable Optical Attenuator

Figure 11. Classical to Quantum Module.

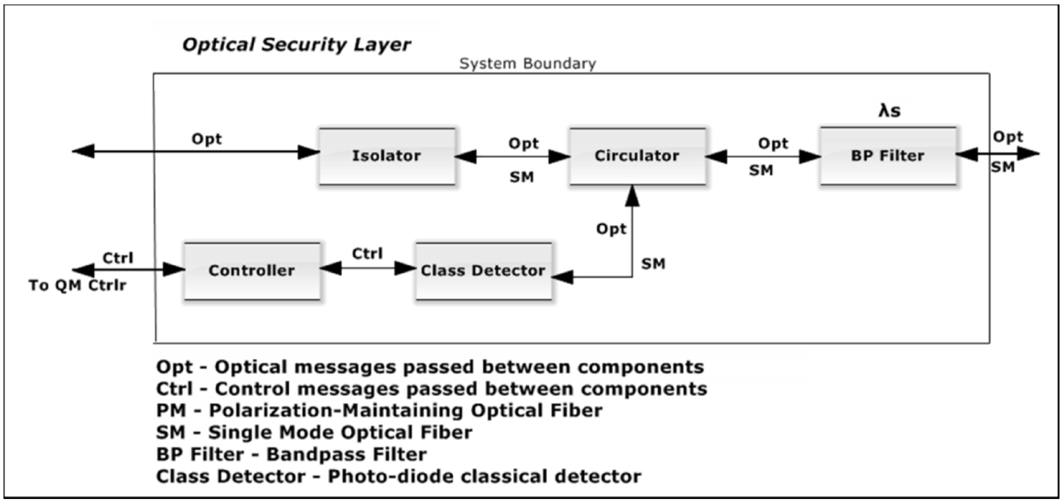

Optical Security Layer
System Boundary
λs
Opt
Isolator
Opt
SM
Circulator
Opt
SM
BP Filter
Opt
SM
Opt
SM
Ctrl
To QM Ctrlr
Controller
Ctrl
Class Detector
Opt - Optical messages passed between components
Ctrl - Control messages passed between components
PM - Polarization-Maintaining Optical Fiber
SM - Single Mode Optical Fiber
BP Filter - Bandpass Filter
Class Detector - Photo-diode classical detector

Figure 12. Optical Security Layer Module.

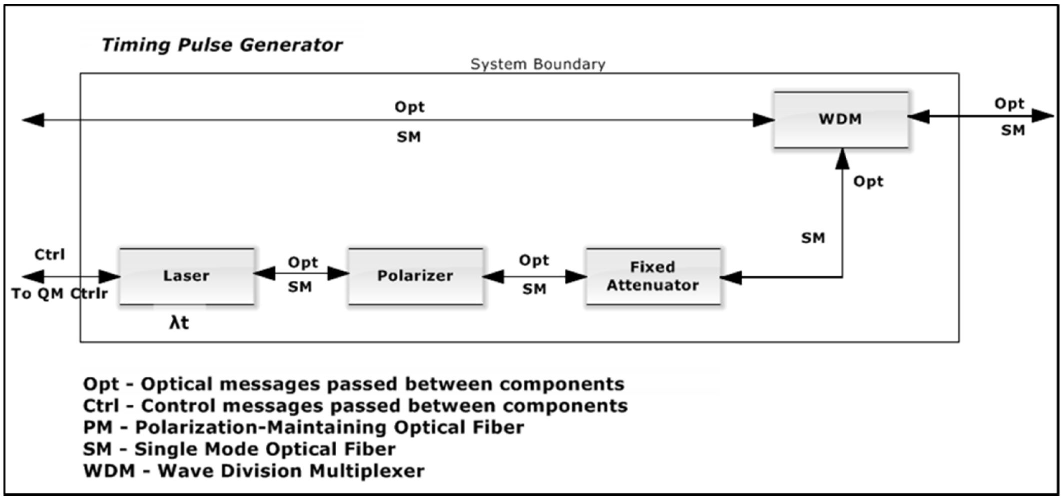

Timing Pulse Generator
System Boundary
Opt
SM
WDM
Opt
SM
Opt
SM
Ctrl
To QM Ctrlr
Laser
λt
Opt
SM
Polarizer
Opt
SM
Fixed Attenuator
Opt - Optical messages passed between components
Ctrl - Control messages passed between components
PM - Polarization-Maintaining Optical Fiber
SM - Single Mode Optical Fiber
WDM - Wave Division Multiplexer

Figure 13. Timing Pulse Generator Module.

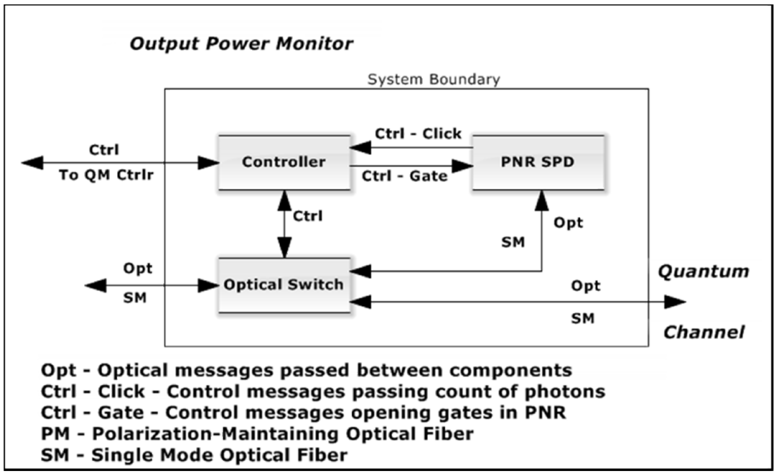

Output Power Monitor
System Boundary
Ctrl
To QM Ctrlr
Controller
Ctrl - Click
Ctrl - Gate
PNR SPD
Ctrl
Opt
SM
Opt
SM
Optical Switch
Opt
SM
Quantum
Channel
Opt - Optical messages passed between components
Ctrl - Click - Control messages passing count of photons
Ctrl - Gate - Control messages opening gates in PNR
PM - Polarization-Maintaining Optical Fiber
SM - Single Mode Optical Fiber

Figure 14. Output Power Monitor Module.

A.1.8 Activity to Systems Function Traceability Matrices

The first table lists the systems functions identified for the BB84 QKD system on the left and shows which Activities (See Fig. 2) each function supports. The areas highlighted in yellow are the focus of this research.

Table 2. *Activity to Systems Function Matrix*.

System Function	**Operational Activity**	Conduct Power-on Tests	Perform Loopback Test	Establish Ethernet Comms	Conduct Authentication Routines	Local Calibration	Global Calibration	**Prepare Qubits**	**Send Qubits**	**Measure Qubits**	Communicate Detection Bases	Discard Disagreeing Bases	Analyze Classical Error	Analyze Quantum Error	Segment Sifted Buffer	Process Segments	Calculate Entropy	Transform Buffer	Reduce Buffer	Hash Final Key	Compare Final Key Hash	Distribute Final Key
Check power		X																				
Check basic parameters		X																				
Check Ethernet controller			X																			
Check quantum module			X																			
Check RNG			X																			
Check System ctrl			X																			

Ethernet packet creation				X	X		X				X					X		X		X	X	X
Ethernet packet sending				X	X		X				X					X		X		X	X	X
Ethernet packet receiving				X	X		X				X					X		X		X	X	X
Authentication routine					X																	
Measure internal environment						X																
Adjust parameters						X	X															
Measure quantum channel							X															
Generate photon								X														
Modulate photon								X														
Attenuate photon								X														
Store qubit data								X		X												
Insert timing pulse									X													
Check output power									X													
Gate detectors										X												
Record detections										X												
Retrieve qubit data											X	X										
Compare qubit data												X										
Discard qubit data												X										
Compare bit subset													X									
Discard bit subset													X									
Chose bit subset													X					X				
Calculate QBER													X									
Calculate stats														X								
Compare stats														X								
Chose block size															X							
Segment sifted key															X							
EC routine																X						
Calculate entropy																	X					
Chose hash function																			X	X		
Compute matrix																			X	X		
Select authentication seed																			X			
Compare hash																					X	

Distribute final key																						X
Distribute authentication seed																						X

This table shows Alice and Bob's subsystems on the left and the Activities at top, showing which subsystem supports each of the Operational Activities from Fig. 2.

Table 3. *Subsystem to Activity Matrix.*

System	**Operational Activity**	Conduct Power-on Tests	Perform Loopback Test	Establish Ethernet Comms	Conduct Authentication	Local Calibration	Global Calibration	**Prepare Qubits**	**Send Qubits**	**Measure Qubits**	Communicate Detection	Discard Disagreeing Bases	Analyze Classical Error	Analyze Quantum Error	Segment Sifted Buffer	Process Segments	Calculate Entropy	Transform Buffer	Reduce Buffer	Hash Final Key	Compare Final Key Hash	Distribute Final Key
Alice System Controller Module		X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X
Alice Public Channel Controller		X	X	X	X	X	X				X					X		X			X	X
Alice QRNG		X	X			X	X	X														
Alice Quantum Module		X	X			X	X	X	X													
Alice Bulk Encryptor																						X
Bob System Controller Module		X	X	X	X	X	X			X	X	X				X		X	X	X	X	X
Bob Public Channel Controller		X	X	X	X	X	X				X					X		X			X	X
Bob QRNG		X	X			X																
Bob Quantum Module		X	X			X	X			X												
Bob Bulk Encryptor																						X

A.2 Selected Research Tools

A.2.1 Software Ideas Modeler

Software Ideas Modeler (http://www.softwareideas.net/) (SIM) is a lightweight Computer-Aided Software Engineering (CASE) tool that provides UML, BPMN, SysML, ERD, Flowcharts, Data Flow Diagrams, Entity Relationship Diagram (Crow Foot/Chen), CRC Cards, User Interface, Hierarchical Task Analysis, Entity Life History, Robustness Diagram, Concurrency Diagram, Venn Diagrams, and Mind Map tools. All of the diagrams for this research were created in SIM.

SIM was chosen as the CASE tool over the more extensive system engineering tools such as Sparx System's Enterprise Architect (http://www.sparxsystems.com/products/ea/) or Vitech's Core (http://www.vitechcorp.com/products/core.shtml) due to either products lack of support for the DEVS. Such tools are ultimately designed to provide support for rules-based modeling and even support simulation. Since neither could provide support for DEVS rules, which necessitated the use of MS4ME to simulate DEVS models, a smaller program with less unnecessary functions was deemed acceptable. SIM provided all of the tools necessary to create the various products and provided linking and traceability between various system engineering products. See Figure 15 for a screen of SIM showing the project overview screen.

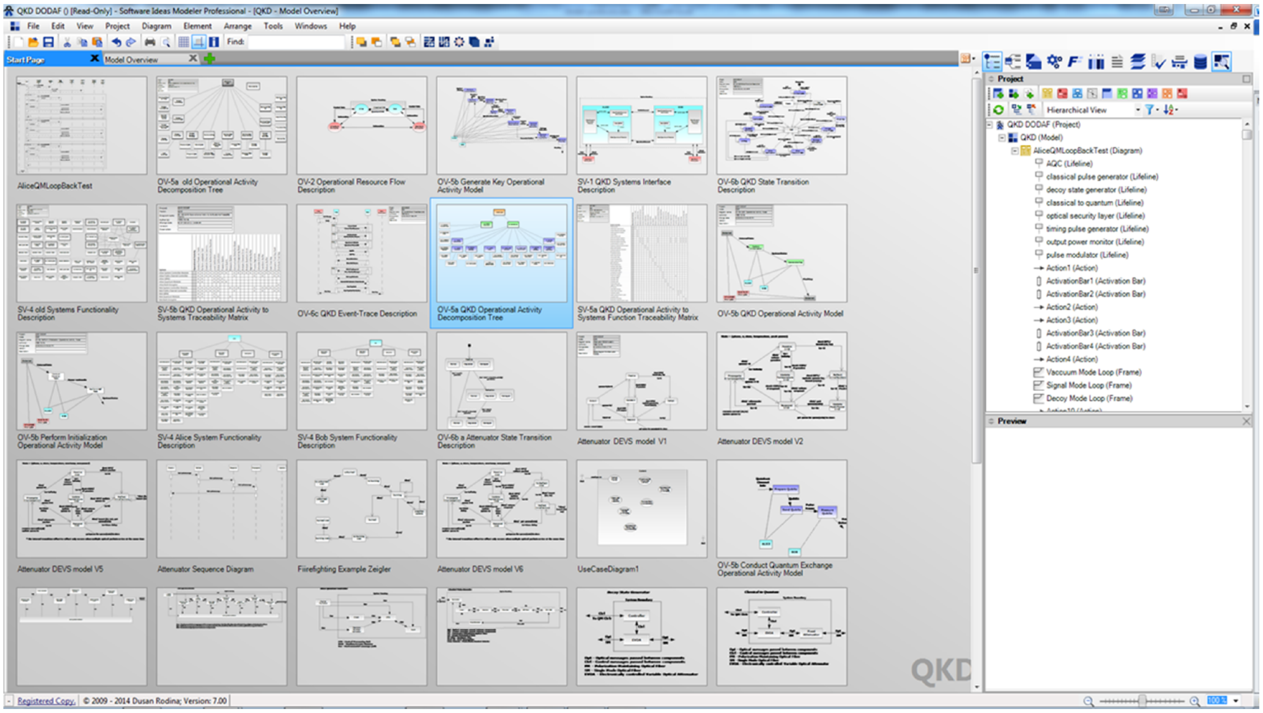
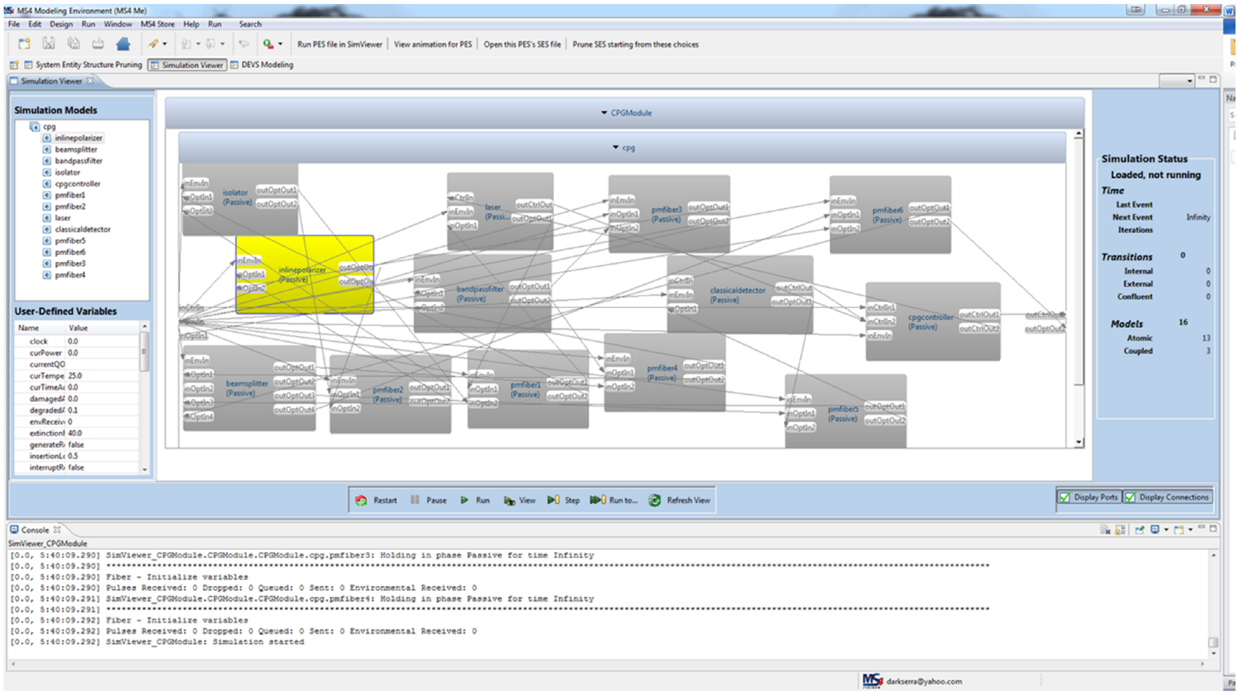

QKD DODAF () [Read-Only] - Software Ideas Modeler Professional - [QKD - Model Overview]
File
Edit
View
Project
Diagram
Element
Arrange
Tools
Windows
Help
Find:
Start Page
Model Overview
AliceQMLoopBackTest
OV-5a old Operational Activity Decomposition Tree
OV-2 Operational Resource Flow Description
OV-5b Generate Key Operational Activity Model
SV-1 QKD Systems Interface Description
OV-6b QKD State Transition Description
SV-4 old Systems Functionality Description
SV-5b QKD Operational Activity to Systems Traceability Matrix
OV-6c QKD Event-Trace Description
OV-5a QKD Operational Activity Decomposition Tree
SV-5a QKD Operational Activity to Systems Function Traceability Matrix
OV-5b QKD Operational Activity Model
OV-5b Perform Initialization Operational Activity Model
SV-4 Alice System Functionality Description
SV-4 Bob System Functionality Description
OV-6b a Attenuator State Transition Description
Attenuator DEVS model V1
Attenuator DEVS model V2
Attenuator DEVS model V5
Attenuator Sequence Diagram
Fiirefighting Example Zeigler
Attenuator DEVS model V6
UseCaseDiagram1
OV-5b Conduct Quantum Exchange Operational Activity Model
QKD
Project
Hierarchical View
QKD DODAF (Project)
QKD (Model)
AliceQMLoopBackTest (Diagram)
AQC (Lifeline)
classical pulse generator (Lifeline)
decoy state generator (Lifeline)
classical to quantum (Lifeline)
optical security layer (Lifeline)
timing pulse generator (Lifeline)
output power monitor (Lifeline)
pulse modulator (Lifeline)
Action1 (Action)
ActivationBar1 (Activation Bar)
ActivationBar2 (Activation Bar)
Action2 (Action)
Action3 (Action)
ActivationBar3 (Activation Bar)
ActivationBar4 (Activation Bar)
Action4 (Action)
Vaccuum Mode Loop (Frame)
Signal Mode Loop (Frame)
Decoy Mode Loop (Frame)
Preview
Registered Copy.
© 2009 - 2014 Dusan Rodina; Version: 7.00

Figure 15. Screenshot of Software Ideas Modeler.

A.2.2 MS4ME

MS4ME (MS4 Systems, 2014). MS4ME is a requirements engineering, data engineering and modeling and simulation tool (http://www.ms4systems.com/pages/ms4me.php), product of RTSync (www.rtsync.com), a spin-off from the Arizona Center of Integrative Modeling and Simulation (ACIMS) (Arizona State University, 2014). MS4ME provides a structured user interface for modeling built on top of the DEVSJAVA simulator (Zeigler, Sarjoughian, & Au, 1997).

This software is the professional successor to the DEVS-Suite simulator built by students and faculty of ACIMS. It uses the DEVSJAVA simulator as a base to model with the DEVS formalism and uses the Eclipse (www.eclipse.org) Integrated Development Environment (IDE) as the user interface. MS4ME provides a simulation environment using the DEVS formalisms, specifically the Finite-Deterministic DEVS (FD-DEVS) and Parallel-DEVS of DEVS. This research used Parallel DEVS to model the optical components, as discussed in Chapter 6. See Fig. 16 for screen capture of MS4ME simulator function running the CPG module.



User-Defined Variables

Name	Value
clock	0.0
curPower	0.0
currentQO	
curTempe	25.0
curTimeAc	0.0
damagedA	0.0
degradedA	0.1
envReceiv	0
extinctionI	40.0
generateR	false
insertionLc	0.5
interruptR	false

Simulation Status
Loaded, not running
Time
Last Event
Next Event Infinity
Iterations
Transitions 0
Internal 0
External 0
Confluent 0
Models 16
Atomic 13
Coupled 3

SimViewer_CPGModule

```
[0.0, 5:40:09.290] SimViewer_CPGModule.CPGModule.CPGModule.cpg.pmfiber3: Holding in phase Passive for time Infinity
[0.0, 5:40:09.290] ***************************************************************************************************************************************************************************
[0.0, 5:40:09.290] Fiber - Initialize variables
[0.0, 5:40:09.290] Pulses Received: 0 Dropped: 0 Queued: 0 Sent: 0 Environmental Received: 0
[0.0, 5:40:09.291] SimViewer_CPGModule.CPGModule.CPGModule.cpg.pmfiber4: Holding in phase Passive for time Infinity
[0.0, 5:40:09.291] ***************************************************************************************************************************************************************************
[0.0, 5:40:09.292] Fiber - Initialize variables
[0.0, 5:40:09.292] Pulses Received: 0 Dropped: 0 Queued: 0 Sent: 0 Environmental Received: 0
[0.0, 5:40:09.292] SimViewer_CPGModule: Simulation started
```

Figure 16. MS4ME Screenshot.

A.3 References

Arizona State University. (2014). Arizona center for integrative modeling and simulation. Retrieved, 2014, Retrieved from http://acims.asu.edu/

Department of Defense. (2010). DoDAF viewpoints and models operational viewpoint AV-1: Overview and summary information. Retrieved, 2012, Retrieved from http://dodcio.defense.gov/dodaf20/dodaf20_av1.aspx

MS4 Systems. (2014). MS4ME. Retrieved, 2014, Retrieved from http://www.ms4systems.com/pages/ms4me.php

Zeigler, B. P., Sarjoughian, H. S., & Au, V. (1997). Object-oriented DEVS: Object behavior specification . Paper presented at the *Proceedings of Enabling Technology for Simulation Science,* Orlando, Fl. *, 3083* 100-111.

Appendix B- Model Creation and Testing Methodology

B.1 Atomic and Coupled Model Creation

As explained in Chapters 2 and 6, the process for creating each coupled and atomic model consisted of many steps. The general process for the optical components started with the optical Subject Matter Expert (SME) creating a mathematical model of the component, using that model to create Discrete Event System Specification (DEVS) pseudocode to capture the behavior and timing aspects, and finally creating MS4ME models using the DEVS pseudocode. These models underwent testing within the MS4ME simulator, with the results shared with the SME for face validity and trace checking (see chapter 6 for explanation of these validation techniques).

Section B.2 describes the component behavior testing shows an example of the MS4ME output from a test case used for the Electronically Controlled Variable Optical Attenuator (EVOA), and lists pseudocode derived from code comments win the EVOA MS4ME code. Each component was designed using the process described below. Creation and testing process steps were:

- Component description – describing the component function and physical design using commercial and academic literature.
- Component conceptual model – text description of the properties and behaviors of interest in the component.
- English-language rules – list of rules that describe the behavior of the component.
- Phase transition diagram – diagram that shows how the component moves from phase to phase within each state. Chapter 6 describes this diagram in detail.
- Event trace diagram – diagrams and tables describing how the component moves from phase to phase for several test cases.
- Use case – creating use cases for the component.
- DEVS code – DEVS pseudocode for the component; used to create the MS4ME models.
- MS4ME code – programming the pseudocode into the MS4ME simulator as a check to ensure the pseudocode captured the behavior and timing properly.

- MS4ME output review – a line-by-line check of the MS4ME output to ensure the MS4ME model and the pseudocode matched and the MS4ME programming was correct.
- MS4ME model review – the optical SME reviewed the DEVS conceptual model and the MS4ME models to ensure the modeled behavior and timing was correct in comparison to the starting mathematical model.
- Model refactor – after review, feedback from the optical SME helped correct each model.

Appendices D-T contains the first seven steps for each optical component. The optical components created for the QKD prototypical architecture were:

Table 4. *Modeled optical components.*

Appendix #	Title	Contents
D	Bandpass Filter	Component description, DEVS documentation & Use Cases
E	Beamsplitter	Component description, DEVS documentation & Use Cases
F	Circulator	Component description, DEVS documentation & Use Cases
G	Photo-diode	Component description, DEVS documentation & Use Cases
H	EVOA	Component description, DEVS documentation & Use Cases
I	Fixed Attenuator	Component description, DEVS documentation & Use Cases
J	Half-wave Plate	Component description, DEVS documentation & Use Cases
K	In-line Polarizer	Component description, DEVS documentation & Use Cases
L	Isolator	Component description, DEVS documentation & Use Cases
M	Laser	Component description, DEVS documentation & Use Cases
N	PM Fiber	Component description, DEVS documentation & Use Cases
O	Polarization Controller	Component description, DEVS documentation & Use Cases
P	Pulse Modulator	Component description, DEVS documentation & Use Cases
Q	Polarizing Beamsplitter	Component description, DEVS documentation & Use Cases
R	SM Fiber	Component description, DEVS documentation & Use Cases

S	Optical Switch	Component description, DEVS documentation & Use Cases
T	WDM	Component description, DEVS documentation & Use Cases

The process for coupled models (joining atomic models together) followed the same design, except there was no starting mathematical model for the coupled submodules; therefore there was no need for a final comparison to a starting mathematical model by the optical SME. Appendices W-AA contains the documentation for each of the coupled submodules.

Table 5 lists the coupled submodules created for the QKD prototypical architecture.

Table 5. *Modeled Alice submodules components.*

Appendix #	Title	Contents
U	CPG Module	Submodule description, DEVS documentation & Use Cases
V	PM Module	Submodule description, DEVS documentation & Use Cases
W	DSG Module	Submodule description, DEVS documentation & Use Cases
X	CTQ Module	Submodule description, DEVS documentation & Use Cases
Y	OSL Module	Submodule description, DEVS documentation & Use Cases
Z	TPG Module	Submodule description, DEVS documentation & Use Cases
AA	OPM Module	Submodule description, DEVS documentation & Use Cases

B.2 Component Behavior Testing

After building each component in MS4ME, it was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. This is the validation technique of *traces* (behavior is followed through the model) (Sargent, 2005) . Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly and in different combinations of ports and input messages. After identifying and correcting any errors, the component was retested until it performed correctly for each test case. The component documentation and results went to the optical SME for review to ensure the proper behavior had been captured (the validation technique of *face validity* (Sargent, 2005)).

Table 6. *Summary of component behavior testing.* captures these tests by listing the component on the left and the number and test types across the top. Each component will be in either the Passive or Respond phase when reacting to inputs (see each appendix for the phase transition diagram showing the phases), as noted at the top of the table. The number of tests exercising the particular port type is in the boxes. The first column lists the total number of tests, the following columns lists the number of tests exercising a particular port (optical, ctrl, or env) and how many tests are single or multi-port, and finally the number of math-specific tests. These optical SME created these math tests to exercise the demonstration QKD simulation built earlier and but were included in the MS4ME code for potential future work in comparing the conceptual models to the *qkdX* framework (see chapter 7 for a discussion on this framework).

Table 6. *Summary of component behavior testing.*

	Passive Phase								Respond Phase						
	total tests	optical ports	ctrl port	env port	single port	multiple port	math tests		total tests	optical ports	ctrl port	env port	single port	multiple port	math tests
Bandpass Filter	21	20	0	13	5	16	4		21	21	0	13	4	17	0
Beamsplitter	33	28	0	21	9	24	6		33	33	0	21	8	25	0
Circulator	27	26	0	17	6	21	6		27	27	0	17	6	21	0
Classical Detector	21	14	14	13	5	16	7		21	21	14	13	1	20	0
EVOA	30	24	13	18	6	24	7		22	22	9	11	2	19	0
Fixed Attenuator	21	20	0	13	5	16	7		21	21	0	13	4	17	0
Halfwave plate	21	20	0	13	5	16	8		21	21	0	13	4	17	0
Inline Polarizer	29	20	0	13	5	16	7		21	21	0	13	4	17	0
Isolator	29	20	0	13	5	16	7		21	21	0	13	4	17	0
Laser	21	14	14	13	5	16	7		21	21	15	13	2	19	0
Optical Switch	35	29	14	19	7	28	8		27	27	10	12	2	25	0
PM Fiber	21	20	0	13	3	18	7		21	21	0	13	4	17	0
Polarization Controller	30	24	13	18	6	24	8		22	22	9	11	2	20	0

Polarization Modulator	30	24	13	18	6	24	8		22	22	9	11	2	20	0
Polarizing Beamsplitter	33	28	0	21	9	24	7		33	33	0	21	8	25	0
SM Fiber	21	20	0	13	3	18	7		21	21	0	13	4	17	0
Wave Division Multiplexer	27	26	0	17	4	23	7		27	27	0	17	6	21	0

Table 7. ***Example test case timing chart - EVOA.***shows the *case timing chart* for the EVOA component. The first ten columns of this chart list the test cases, and the last five show the sequence of test cases and subcases during the testing. The chart has a numbered list of test cases on the left by type (Passive, Respond, or Math) and information about each test case across the top. Each case shows which port it is exercising, the numbers in the cells show many messages sent to each port during the test case, and finally a note about timing of the messages. The messages can be a single message to a port, messages that arrive at the same time, or messages that arrive at different times. The last three columns for the test cases show the running total of each message type sent to the component during the test. The bottom of the columns show total message numbers and a notes section for the test cases, any of which are highlighted in orange.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

For example, Test Case 26 starts the component in the Passive phase and injects two messages to port optical 1 (Opt1), one message to optical port 2 (Opt2), one message each to the control and environmental ports, all done at the same time. The intent of the case is to see if the component responds correctly to external inputs while it is processing optical packets. The first optical packets when the component is in Passive, forcing a change of phase. While processing the packet, the other three packets arrive simultaneiously. This causes the component to interrupt the current process and respond by evaluating the incoming packet. After the response, the component needs to restart processing the first packet while updating the time remaining its processing. This component phase transition chart shows this behavior.

At the end of the case, 32 optical, 11 environmental and ten control messages have been sent to the component. As with all test cases, the component showed the proper behavior at the conclusion of the case verified by tracing. Any errors found during testing were fixed and the component retested until passing all test cases.

Table 7. *Example test case timing chart - EVOA.*

		Inject Ports					**Running Totals**		
Phase	**Case**	**Opt1**	**Opt2**	**Ctrl**	**Env**	**Timing**	**opt #**	**env #**	**ctrl #**
Passive	1	1	0	0	0	single	1	0	0
	2	0	1	0	0	single	2	0	0
	3	0	0	1	0	single	2	0	1
	4	0	0	0	1	single	2	1	1
	5	1	1	0	0	same time	4	1	1
	6	1	0	1	0	same time	5	1	2
	7	1	1	0	0	differ time	7	1	2
	8	1	0	1	0	differ time	8	1	3
	9	1	1	1	1	same time	10	2	4
	10	1	1	1	1	differ time	12	3	5
	11	0	1	0	1	same time	13	4	5
	12	0	1	0	1	differ time	14	5	5
	13	0	0	1	1	same time	14	6	6
	14	0	0	1	1	differ time	14	7	7
	15	1	0	0	1	same time	15	8	7
	16	1	0	0	1	differ time	16	9	7
	20	2	0	0	0	same time	18	9	7
	21	0	2	0	0	same time	20	9	7
	22	2	1	0	0	same time	23	9	7

	23	2	0	1	0	same time	25	9	8
	24	2	0	0	1	same time	27	10	8
	25	2	0	1	0	differ time	29	10	9
	26	2	1	1	1	same time	32	11	10
	27	2	1	1	1	differ time	35	12	11
	28	0	2	0	1	same time	37	13	11
	29	0	2	0	1	differ time	39	14	11
	30	0	0	1	1	same time	39	15	12
	31	0	0	1	1	differ time	39	16	13
	32	2	0	0	1	same time	41	17	13
	33	2	0	0	1	differ time	43	18	13
totals		27	16	13	18				
Respond	41	2	0	0	0	single	45	18	13
	42	1	1	0	0	single	47	18	13
	43	1	0	1	0	single	48	18	14
	44	1	0	0	1	single	49	19	14
	45	2	1	0	0	same time	52	19	14
	46	2	0	1	0	same time	54	19	15
	47	2	0	0	1	differ time	56	20	15
	48	2	0	1	0	differ time	58	20	16
	49	2	1	1	1	same time	61	21	17
	50	2	1	1	1	differ time	64	22	18
	51	1	1	0	1	same time	66	23	18
	52	1	1	0	1	differ time	68	24	18
	60	3	0	0	0	same time	71	24	18
	61	1	2	0	0	same time	74	24	18
	62	3	1	0	0	same time	78	24	18
	63	3	0	1	0	same time	81	24	19
	64	3	0	0	1	same time	84	25	19
	65	3	0	1	0	differ time	87	25	20
	66	3	1	1	1	same time	91	26	21
	67	3	1	1	1	differ time	95	27	22
	68	1	2	0	1	same time	98	28	22
	69	1	2	0	1	differ time	101	29	22
totals		43	15	9	11				
Math	TC1	1	0	1	2	same time	102	31	23
	TC2	1	0	1	2	same time	103	33	24
	TC3	1	0	1	2	same time	104	35	25
	TC4	1	0	1	2	same time	105	37	26
	TC5	1	0	1	2	same time	106	39	27
	TC6	1	0	1	2	same time	107	41	28
	TC7	1	0	0	2	same time	108	43	28

totals	7	0	6	14
totals	77	31	28	43

Notes:

8 - Set attenuation message, set newattenuation = 2
10 - Get attenuation message
13 - Increase attenuation message
14 - Decrease attenuation message
23 - INIT control message sent; OPT1 & Ctrl - same time - Passive: downstream received packets = 214
30 - INIT control message sent - Ctrl & ENV - same time - Passive: downstream received packets = 214
63 - INIT control message sent - OPT1 & Ctrl - same time - Respond: downstream received packets = 211
65 - Set attenuation message, set newattenuation = 5
67 - INIT control message sent - OPT1, OPT2, Ctrl & ENV - differ time - Respond: downstream received packets = 207

B.2.1 MS4ME Testing Structure

The testing construct for each component and coupled module is the same. There is test component called "Upstream" that injects messages into the component under test and there is a testing component called "Downstream" that captures all output from the component under test. These two constructs contain all of the logic for the testing cases, as the component only has logic to react to inputs. Figure 17 shows the *testevoa* module with the three components, each component labeled with its current phase and ports. As noted earlier, Upstream only has (out)put ports connected to the EVOA (in)put ports, and all of the EVOA (out)put ports are connected to Downstream's (in)put ports. Since the EVOA does not output env(ironmental) messages, there are is no env input port in Downstream. Each optical component listed in Table 4 has a similar testing construct built in MS4ME.

.

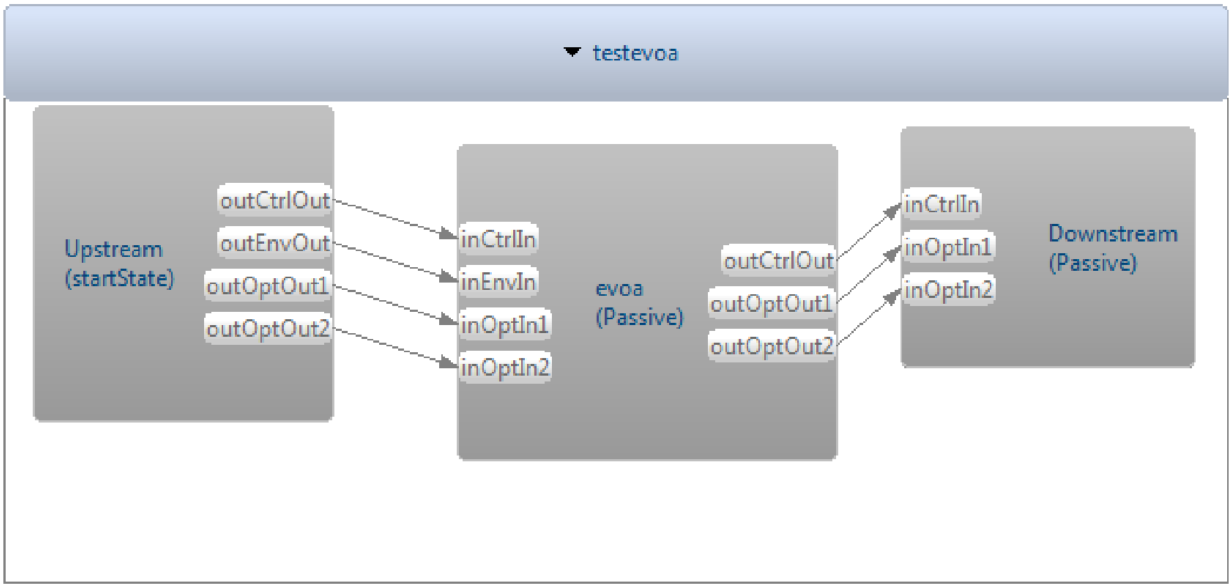


Figure 17. EVOA testing structure in MS4ME

B.2.2 MS4ME Sample EVOA Output

Following this paragraph is output from MS4ME while running the EVOA component. This is the output from Test Case 2 (highlighted in Table 6) and starts at simulation time 20. Major events described in this paragraph are highlighted in yellow in the MS4ME output. An optical pulse with id number 2 enters the EVOA at simulation time 21.0. The attenuator reacts to the input with an external event during the Passive phase on optical port 2 (OptIn2). The packet is added to the queue and the attenuator transitions from Passive to Reflect phase. The component reflects a portion of the packet out to the Downstream module and enters the Reflect internal transition. During the Reflect phase internal transition, the queued packet is removed from the queue, attenuated, and set to output during the Respond output phase. Once in the Respond phase, the packet is output and the queue checked during the Respond internal transition. Since the queue contains zero packets, the EVOA returns to the Passive phase and finally, Downstream receives the attenuated optical packet at simulation time 26. This time checks properly as the propagation time for the EVOA is set to five for this test case (26-21=5).

```
[20.0, 1:54:00.675] SimViewer_testevoa: Simulation step started
[20.0, 1:54:00.675] Upstream - output event for Wait1 - time Elapsed: NA getTimeAdvance: 9.0 clock: 20.0
[20.0, 1:54:00.676] SimViewer_testevoa.testevoa.testevoa.Upstream: Internal transition
[20.0, 1:54:00.676] SimViewer_testevoa.testevoa.testevoa.Upstream: Internal transition from Wait1
[20.0, 1:54:00.677] SimViewer_testevoa.testevoa.testevoa.Upstream: Holding in phase Case2 for time 0.0
[20.0, 1:54:00.678] SimViewer_testevoa: Simulation step finished
[20.0, 1:54:00.685] SimViewer_testevoa: Simulation step started
[20.0, 1:54:00.685] Upstream - output event for Case2 - time Elapsed: NA getTimeAdvance: 0.0 clock: 20.0
[20.0, 1:54:00.686] SimViewer_testevoa.testevoa.testevoa.Upstream: Internal transition
[20.0, 1:54:00.686] SimViewer_testevoa.testevoa.testevoa.Upstream: Internal transition from Case2
[20.0, 1:54:00.686] SimViewer_testevoa.testevoa.testevoa.Upstream: Holding in phase Case2_1 for time 1.0
[20.0, 1:54:00.686] Upstream - internal event for Case2 - time Elapsed: NA getTimeAdvance: 1.0 clock: 20.0
[20.0, 1:54:00.686] Upstream - internal event for Case2 - time Elapsed: NA getTimeAdvance: 0.0 clock: 20.0
[20.0, 1:54:00.687] SimViewer_testevoa: Simulation step finished
[21.0, 1:54:00.703] SimViewer_testevoa: Simulation step started
[21.0, 1:54:00.704] Upstream - output event for Case2_1 - time Elapsed: NA getTimeAdvance: 1.0 clock: 21.0
[21.0, 1:54:00.704] SimViewer_testevoa.testevoa.testevoa.Upstream: Internal transition
[21.0, 1:54:00.704] SimViewer_testevoa.testevoa.testevoa.Upstream: Internal transition from Case2_1
[21.0, 1:54:00.705] SimViewer_testevoa.testevoa.testevoa.Upstream: Holding in phase Wait2 for time 9.0
[21.0, 1:54:00.705] Upstream - internal event for Case2_1 - time Elapsed: NA getTimeAdvance: 9.0 clock: 21.0
[21.0, 1:54:00.705] Upstream - internal event for Case2_1 - time Elapsed: NA getTimeAdvance: 1.0 clock: 21.0
[21.0, 1:54:00.705] SimViewer_testevoa.testevoa.testevoa.evoa: Received messages [inOptIn2: OpticalPulse
            id: 2
            name: Case 2 Optical Pulse 2
            duration: 4.0E-10
            opticalPower: 8.709666395E-5
            brightPulseFlg: false
            amplitude: 1.94095418E-6
            centralFrequency: 1.0E15
            globalPhase: 0.0
            ellipticity: 0.0
            orientation: 0.0
            numberOfGaussians: 3
            gA0: 45.5
            gA1: 38.064
            gA2: 4.75752
            gM0: 9.54844E-11
            gM1: 1.81125E-10
            gM2: 3.52322E-10
            gSD0: 1.98151E-11
            gSD1: 5.81389E-11
            gSD2: 4.90169E-11]
[21.0, 1:54:00.706] SimViewer_testevoa.testevoa.testevoa.evoa: External transition
[21.0, 1:54:00.706] SimViewer_testevoa.testevoa.testevoa.evoa: Holding in phase Reflect for time 0.0
[21.0, 1:54:00.706] @@************* EVOA - START PASSIVE OPTIN2 EXTERNAL
EVENT******************************************************************
[21.0, 1:54:00.706] EVOA - external event for Passive with OptIn2 - time Elapsed: 5.0 getTimeAdvance: 0.0 clock: 21.0
[21.0, 1:54:00.706] Pulses Received: 2 Reflected: 1 Queued: 1 Sent: 1 Environmental Received: 0 Control Received: 0
[21.0, 1:54:00.706] EVOA - received optical pulse named : Case 2 Optical Pulse 2 at time : 21.0
[21.0, 1:54:00.706] Passive External - Adding pulse to queue
[21.0, 1:54:00.706] ****************** DISPLAY PULSES IN QUEUE ******************
[21.0, 1:54:00.706] Total Number of Optical Pulses in Queue  : 1
[21.0, 1:54:00.706] ID: 2 Name: Case 2 Optical Pulse 2 PortNum: 2 Reflected: false Time Remaining: 5.0
[21.0, 1:54:00.706] *************************************************************
[21.0, 1:54:00.706] Pulses Received: 2 Reflected: 1 Queued: 2 Sent: 1 Environmental Received: 0 Control Received: 0
[21.0, 1:54:00.706] EVOA - Preparing reflected pulse: Reflected Case 2 Optical Pulse 2 for sending on port OptOut2
[21.0, 1:54:00.706] Reflected pulse ID: 2 Name: Reflected Case 2 Optical Pulse 2 PortNum: 2 Reflected: true Time Remaining: 5.0
[21.0, 1:54:00.706] @@************* EVOA - END PASSIVE OPTIN2 EXTERNAL
EVENT******************************************************************
[21.0, 1:54:00.707] SimViewer_testevoa: Simulation step finished
[21.0, 1:54:00.713] SimViewer_testevoa: Simulation step started
[21.0, 1:54:00.713] @@************* EVOA - START REFLECT OUTPUT
EVENT******************************************************************
[21.0, 1:54:00.713] EVOA - output event for Reflect - time Elapsed: NA getTimeAdvance: 0.0 clock: 21.0
[21.0, 1:54:00.713] ****************** DISPLAY PULSES IN QUEUE ******************
[21.0, 1:54:00.713] Total Number of Optical Pulses in Queue  : 1 before reflection.
[21.0, 1:54:00.713] ID: 2 Name: Case 2 Optical Pulse 2 PortNum: 2 Reflected: true Time Remaining: 5.0
[21.0, 1:54:00.713] *************************************************************
[21.0, 1:54:00.713] ### EVOA - Reflecting pulse: Reflected Case 2 Optical Pulse 2 to port OptOut2
```

```
[21.0, 1:54:00.714] Current Attenuation =: 0.1
[21.0, 1:54:00.714] New Attenuation =: 0.1
[21.0, 1:54:00.714] Pulses Received: 2 Reflected: 2 Queued: 2 Sent: 1 Environmental Received: 0 Control Received: 0
[21.0, 1:54:00.714] @@************* EVOA - END REFLECT OUTPUT
EVENT********************************************************************
[21.0, 1:54:00.714] SimViewer_testevoa.testevoa.testevoa.evoa: Internal transition
[21.0, 1:54:00.715] SimViewer_testevoa.testevoa.testevoa.evoa: Internal transition from Reflect
[21.0, 1:54:00.715] SimViewer_testevoa.testevoa.testevoa.evoa: Holding in phase Respond for time 5.0
[21.0, 1:54:00.715] @@************* EVOA - START REFLECT INTERNAL
EVENT********************************************************************
[21.0, 1:54:00.715] EVOA - output event for Reflect: Total Number of Optical Pulses in Queue  : 1
[21.0, 1:54:00.715] ******************* DISPLAY PULSES IN QUEUE *******************
[21.0, 1:54:00.715] Total Number of Optical Pulses in Queue  : 1 after reflection.
[21.0, 1:54:00.715] ID: 2 Name: Case 2 Optical Pulse 2 PortNum: 2 Reflected: true Time Remaining: 5.0
[21.0, 1:54:00.715] ***************************************************************
[21.0, 1:54:00.715] Setting up the pulse - polling the queue and REMOVING FROM QUEUE
[21.0, 1:54:00.715] ** INTERNAL EVENT FOR RESPOND - Before Calc - Amplitude: 1.94095418E-6
[21.0, 1:54:00.715] ** INTERNAL EVENT FOR RESPOND - After Calc - Amplitude: 1.918736261226321E-6
[21.0, 1:54:00.715] EVOA - Preparing send pulse: Attenuated Case 2 Optical Pulse 2 to port OptOut1
[21.0, 1:54:00.716] SimViewer_testevoa.testevoa.testevoa.evoa: Holding in phase Respond for time 5.0
[21.0, 1:54:00.716] Current Attenuation =: 0.1
[21.0, 1:54:00.716] New Attenuation =: 0.1
[21.0, 1:54:00.716] @@************* EVOA - END REFLECT INTERNAL TRANSITION
*******************************************************************
[21.0, 1:54:00.716] SimViewer_testevoa.testevoa.testevoa.Downstream: Received messages [inOptIn2: OpticalPulse
        id: 2
        name: Reflected Case 2 Optical Pulse 2
        duration: 4.0E-10
        opticalPower: 8.709666395E-5
        brightPulseFlg: false
        amplitude: 1.94095418E-9
        centralFrequency: 1.0E15
        globalPhase: 0.0
        ellipticity: 0.0
        orientation: 0.0
        numberOfGaussians: 3
        gA0: 45.5
        gA1: 38.064
        gA2: 4.75752
        gM0: 9.54844E-11
        gM1: 1.81125E-10
        gM2: 3.52322E-10
        gSD0: 1.98151E-11
        gSD1: 5.81389E-11
        gSD2: 4.90169E-11]
[21.0, 1:54:00.717] SimViewer_testevoa.testevoa.testevoa.Downstream: External transition
[21.0, 1:54:00.717] SimViewer_testevoa.testevoa.testevoa.Downstream: Holding in phase Passive for time Infinity
[21.0, 1:54:00.717] Downstream - external event for Passive with OptIn2 - time Elapsed: 5.0 getTimeAdvance: Infinity clock: 21.0
[21.0, 1:54:00.717] Downstream - Received : 3
[21.0, 1:54:00.717] Downstream - Received Optical Pulse : 3 named : Reflected Case 2 Optical Pulse 2 at time : 21.0
[21.0, 1:54:00.717] SimViewer_testevoa: Simulation step finished
[26.0, 1:54:00.726] SimViewer_testevoa: Simulation step started
[26.0, 1:54:00.727] @@************* EVOA - START RESPOND OUTPUT
EVENT********************************************************************
[26.0, 1:54:00.727] EVOA - output event for Respond - time Elapsed: NA getTimeAdvance: 5.0 clock: 26.0
[26.0, 1:54:00.727] ******************* DISPLAY PULSES IN QUEUE *******************
[26.0, 1:54:00.727] Total Number of Optical Pulses in Queue  : 0 BEFORE propagation.
[26.0, 1:54:00.727] ***************************************************************
[26.0, 1:54:00.727] ** OUTPUT EVENT FOR RESPOND Amplitude: 1.918736261226321E-6
[26.0, 1:54:00.727] ### EVOA - Sending pulse: Attenuated Case 2 Optical Pulse 2 to port OptOut1
[26.0, 1:54:00.727] Current Attenuation =: 0.1
[26.0, 1:54:00.727] New Attenuation =: 0.1
[26.0, 1:54:00.727] Pulses Received: 2 Reflected: 2 Queued: 2 Sent: 2 Environmental Received: 0 Control Received: 0
[26.0, 1:54:00.727] @@************* EVOA - END RESPOND OUTPUT
EVENT********************************************************************
[26.0, 1:54:00.727] SimViewer_testevoa.testevoa.testevoa.evoa: Internal transition
[26.0, 1:54:00.728] SimViewer_testevoa.testevoa.testevoa.evoa: Internal transition from Respond
[26.0, 1:54:00.728] SimViewer_testevoa.testevoa.testevoa.evoa: Holding in phase Passive for time Infinity
[26.0, 1:54:00.728] @@************* EVOA - START RESPOND INTERNAL
EVENT********************************************************************
```

```
[26.0, 1:54:00.728] EVOA - internal event for Respond - time Elapsed: NA getTimeAdvance: Infinity clock: 26.0
[26.0, 1:54:00.728] ******************* DISPLAY PULSES IN QUEUE *******************
[26.0, 1:54:00.728] Total Number of Optical Pulses in Queue  : 0 BEFORE propagation.
[26.0, 1:54:00.728] ***************************************************************
[26.0, 1:54:00.728] *************** Adjust queue ***************
[26.0, 1:54:00.728] Total Number of Optical Pulses in Queue  : 0
[26.0, 1:54:00.728] Pulses Received: 2 Reflected: 2 Queued: 2 Sent: 2 Environmental Received: 0 Control Received: 0
[26.0, 1:54:00.728] EVOA - Ending Respond internal transition
[26.0, 1:54:00.728] Going to Passive Phase Next!!!
[26.0, 1:54:00.729] SimViewer_testevoa.testevoa.testevoa.evoa: Holding in phase Passive for time Infinity
[26.0, 1:54:00.729] Current Attenuation =: 0.1
[26.0, 1:54:00.729] New Attenuation =: 0.1
[26.0, 1:54:00.729] @@************* EVOA - END RESPOND INTERNAL
EVENT*********************************************************************
[26.0, 1:54:00.729] SimViewer_testevoa.testevoa.testevoa.Downstream: Received messages [inOptIn1: OpticalPulse
            id: 2
            name: Attenuated Case 2 Optical Pulse 2
            duration: 4.0E-10
            opticalPower: 8.709666395E-5
            brightPulseFlg: false
            amplitude: 1.918736261226321E-6
            centralFrequency: 1.0E15
            globalPhase: 0.0
            ellipticity: 0.0
            orientation: 0.0
            numberOfGaussians: 3
            gA0: 45.5
            gA1: 38.064
            gA2: 4.75752
            gM0: 9.54844E-11
            gM1: 1.81125E-10
            gM2: 3.52322E-10
            gSD0: 1.98151E-11
            gSD1: 5.81389E-11
            gSD2: 4.90169E-11]
[26.0, 1:54:00.730] SimViewer_testevoa.testevoa.testevoa.Downstream: External transition
[26.0, 1:54:00.730] SimViewer_testevoa.testevoa.testevoa.Downstream: Holding in phase Passive for time Infinity
[26.0, 1:54:00.730] Downstream - external event for Passive with OptIn1 - time Elapsed: 5.0 getTimeAdvance: Infinity clock: 26.0
[26.0, 1:54:00.730] Downstream - Received : 4
[26.0, 1:54:00.730] Downstream - Received Optical Pulse : 4 named : Attenuated Case 2 Optical Pulse 2 at time : 26.0
[26.0, 1:54:00.731] SimViewer_testevoa: Simulation step finished
```

B.2.3 DEVS MS4ME derived pseudocode

This derived pseudocode is an example of the logic and events within a component, and came from the comments placed within the MS4ME EVOA module code, condensing its 2600 lines of code. The comments were placed to identify actions and phase transitions to increase readability of the code and to provide markers as the code executes during runtime. This pseudocode show a major process for the EVOA is to change its current attenuation setting in response to control input, so the EVOA includes an additional phase of “Update Attenuation” for this operation. See appendix H for the detailed description of the EVOA.

```
EVOA - START PASSIVE

EVOA - START PASSIVE EXTERNAL EVENT
  adjust clock based upon time elapsed since last event
  check for existing optical message
    remove each pulse in bag and store it into a queued optical pulse buffer
  check for existing env message
    check if the received temperature exceeds damaged or degraded threshold
  check for existing ctrl message
    generate the response message for an incoming control message
  set up for first reflected pulse if optical pulse arrived
  go to the Reflect phase else go to the Update Attenuation phase

EVOA - START REFLECT OUTPUT EVENT
  adjust clock based upon time advance
  output pulse

EVOA - START REFLECT INTERNAL EVENT
  identify any pulses that have not yet been reflected and reflect them
  set up pulse to send in Respond phase
  go to the Respond phase

EVOA - START UPDATE ATTENUATION EXTERNAL EVENT
  adjust clock based upon time elapsed since last event
  check for existing optical message
    remove each pulse in bag and store it into a queued optical pulse buffer
  check for existing env message
    check if the received temperature exceeds damaged or degraded threshold
  check for existing ctrl message
    generate the response message for an incoming control message
  set up for first reflected pulse if optical pulse arrived
  else hold in the Update Attenuation phase for a change in Attenuation

EVOA - START UPDATE ATTENUATION OUTPUT EVENT
  adjust clock based upon time advance
  output the response message
  change the current attenuation

EVOA - START UPDATE ATTENUATION INTERNAL EVENT
  set up pulse to send in Respond phase if optical pulse arrived
 go to Respond if optical pulse arrived
else go to Update Attenuation if a control message arrived
else go to Passive

 EVOA - START RESPOND EXTERNAL EVENT
  adjust clock based upon time elapsed since last event
  adjust queue
    set a flag that the Respond phase was interrupted by an external event
  check for existing optical message
    remove each pulse in bag and store it into a queued optical pulse buffer
  check for existing env message
    check if the received temperature exceeds damaged or degraded threshold
  check for existing ctrl message
    generate the response message for an incoming control message
  set up for first reflected pulse if optical pulse arrived
  go to the Reflect phase
    else go to the Update Attenuation phase

EVOA - START RESPOND OUTPUT EVENT
  adjust clock based upon time advance
  adjust pulse amplitude if damaged or degraded
  output pulse to the correct port

EVOA - START RESPOND INTERNAL EVENT
  adjust queue
  check if there any pulses remaining in queue
    set up the next queued pulse
    pulses remaining, so go to Respond
  else check to see if the attenuation is changing
    attenuation changing, go to Update Attenuation
  else go to Passive phase

EVOA – END PASSIVE
```

B.3 Coupled Submodules

The component testing design facilitated building the coupled submodules by replacing the Upstream and Downstream test components with other optical components. For the example of the Classical Pulse Generator (CPG) (see appendix U and chapter 6 for discussion of the CPG), the ‘internal’ of the submodule is built of optical components and the ‘external’ construct is once again an Upstream, a Downstream and the CPG acting as an atomic component, in this case the testing construct is labeled “expframe.” This is the benefit of DEVS’ *closure under coupling* (Chow, 1996; Zeigler, 1976; Zeigler, 1984)and the MS4ME simulator models this behavior.

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was to test the ability of the coupled submodule to receive messages and pass them internally to the submodule controller. The controller processed the message and passed the appropriate message to the controlled component. For example, the CPG submodule received a control message to fire the laser. The CPG received the message, passed it internally to the CPG controller, which passed the message to the laser component, which fired the laser to emit an optical packet. There are four or five test cases for each submodule:

- Case 1 – Send a control message to the coupled submodule to exercise the controller and controlled component. For each of the modules:
  - CPG – control message fires laser.
  - PM – control message changes polarization.
  - DSG – control message change attenuation of EVOA.
  - CTQ – control message change attenuation of EVOA.
  - OSL – control message requests status.
  - TPG – control message fires laser.
  - OPM – control message change switch position.
- Case 2 – Send a control message to request the status of the coupled submodule, which is held in the submodule controller.
- Case 3 – Send an environmental message to the coupled submodule for replication to each component within the module.
- Case 4- Send a control message to the coupled submodule that contains a reset command.
- Case 5 – Send an optical message to submodule for injection into optical path within the submodule (for all but the CPG).

The exceptions for the test cases were the CPG and the OSL. The CPG did not have a test case to inject an optical packet as the CPG is the module that creates the optical pulse, though it did receive reflections from the Downstream module into the CPG OptIn2 port. The OSL had one less control message as its “controlled” device is the classical detector, which only sends data to the controller, and does not accept input control messages. See Table 8 for summary of these tests.

Table 8. *Summary of coupled submodule behavior testing.*

	total tests	Test Type optical ports	 ctrl port	 env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

In the following figures, Figure 18 is the CPG architecture diagram, Figure 19 is the high-level test frame and Figure 20 shows the exploded view of the CPG submodule within the high-level test frame. Notice there is a "cpgcontroller" that receives messages from outside the CPG. This is a simple representation of the logic circuits necessary to operate this module. In these conceptual models, the controller reacts to the appropriate message types for the opto-electrical components connected to the controller. For example, the CPG controller accepts messages for the laser and the classical detector. Similarly, each coupled model has controller specific for its needs.

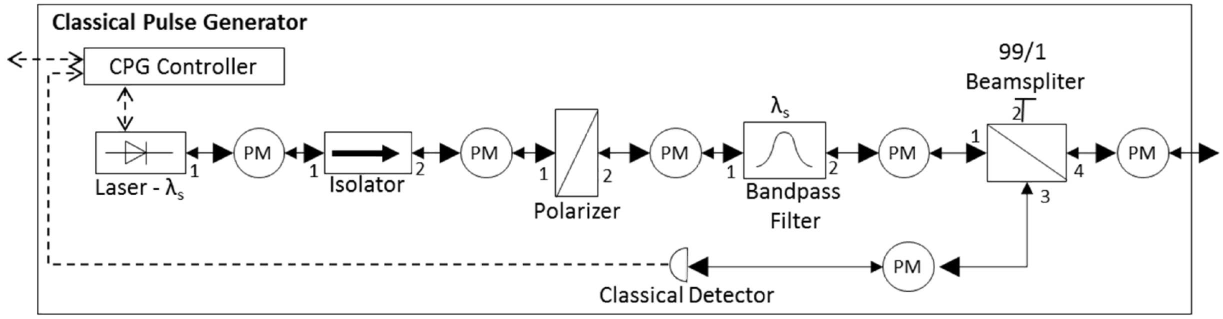


Figure 18. CPG architecture.

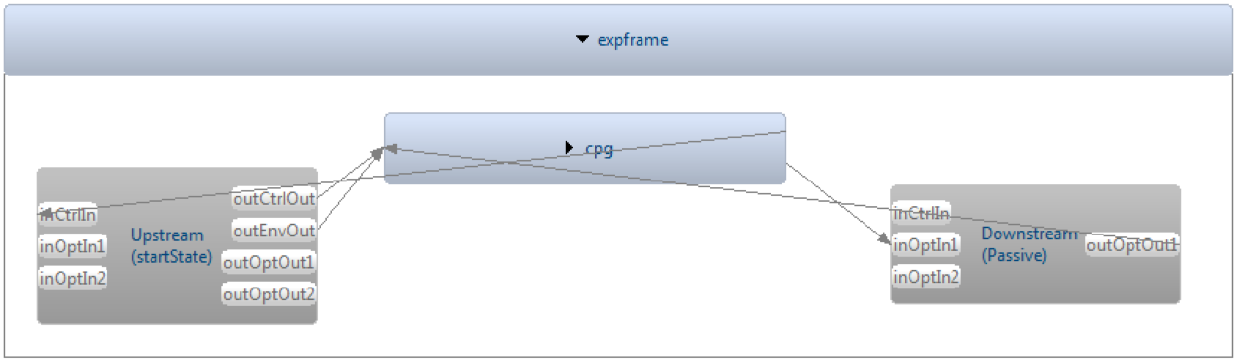


Figure 19. CPG test frame.

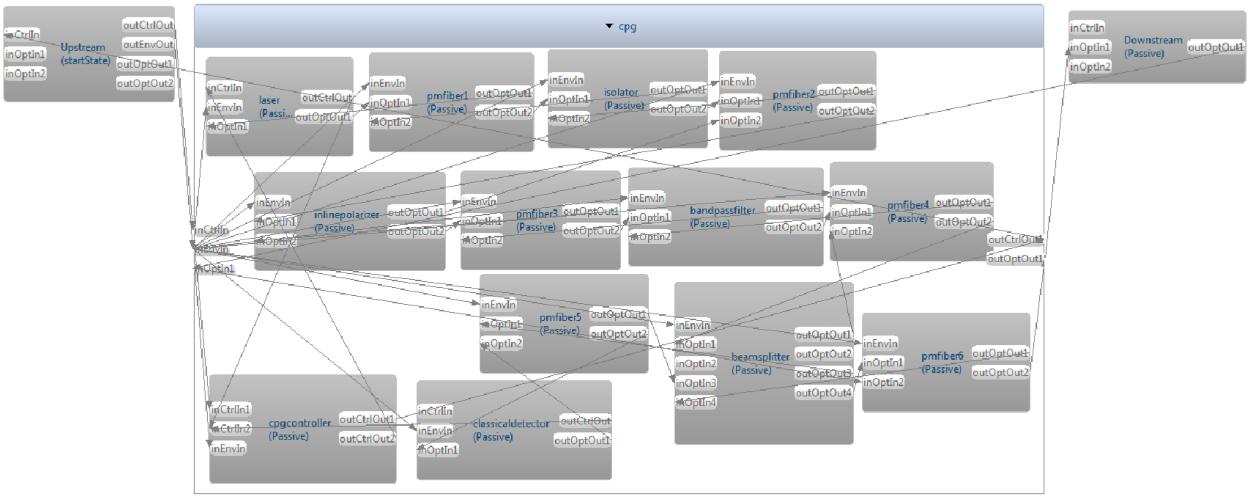


Figure 20. CPG coupled submodule components.

Connections for the coupled models are more complex than for the individual components, as both Upstream and Downstream need send and receive messages to test the external CPG submodule ports, and environmental messages have to pass through into the coupled model for each component.

The code necessary for building coupled models much simpler than the code for the components. The coupled model code specifies external components (Upstream, Downstream) and external inputs and outputs for the CPG module. Additionally, the code lists all internal components and the internal connections between them. Finally, the connections from the CPG to internal components are defined. For a coupled model, there are no phases or transitions defined, as the ‘state’ of the coupled model is the sum of all current states of all internal components. Appendix U describes the CPG in detail and each coupled module appendix (U-AA) has the DEVS code for the controller and the coupled module.

Once constructed, each coupled model was tested by injecting the proper message packet into the submodule. Every coupled model, except the CPG, has input optical, environmental and control ports. The CPG has no input optical port, as the laser within the CPG generates optical pulses for the Alice subsystem and its primary input is a control message to fire the laser. Table 9 summarizes the test cases for the CPG submodule.

Table 9. *Example test case timing chart - CPG.*

	Inject Ports				**Running Totals**		
Case	**Opt1**	**Ctrl**	**Env**	**Timing**	**opt #**	**env #**	**ctrl #**
1	0	1	0	single	0	0	1
2	0	1	0	single	0	0	2
3	0	0	1	single	0	1	2
4	0	1	0	single	0	1	3
totals	0	3	1				

B.3.1 MS4ME Sample CPG Output

This section contains the output from MS4ME while running the CPG submodule. This is the output from Test Case 1 (highlighted in Table 9) and starts at simulation time 10. Major events described in this paragraph are highlighted in yellow in the following MS4ME output. Case 1 starts with Upstream having an output event at time 10, sending the message to the CPG which seamlessly passes the message inside the submodule to the CPG controller. The controller receives a LaserMsg with id:1 on port CtrlIn1 and starts the Passive external event. The controller sees it has received a laser fire message and moves to the Respond phase. The controller notes that the status number of the message is 6, the stored power value of the last classical detection is zero (this is sent from the classical detector when it has a detection) and prepares a response message and sends it out port CtlrOut2, ending the Response phase.

The response message goes to the laser, which receives the message on port CtrlIn and starts the Passive CtrlIn event. The laser decodes the message, moves to the "ON" setting and prepares a fire control message for port CtrlOut. The laser moves to the Update Laser phase where it notes the laserpower variable is "true," indicating the laser is on. The laser begins preparing an optical pulse for output on OptOut1 (the only optical output port in the laser) and notes it is moving to the Create Pulse phase. Once in the Create Pulse phase, the laser outputs the generated optical pulse on port OptOut1 and ends the Create Pulse phase. Finally, the next component in line from the laser, the pmfiber1, receives the optical pulse at simulation time 16. This output shows how the CPG, the CPG controller, and the CPG internal laser reacts to a "fire laser" message to generate optical pulses at the beginning of the optical path within Alice.

```
[10.0, 2:13:36.632] SimViewer_expframe: Simulation step started
[10.0, 2:13:36.632] Upstream - output event for Case1 - time Elapsed: NA getTimeAdvance: 10.0 clock: 10.0
[10.0, 2:13:36.632] SimViewer_expframe.expframe.expframe.Upstream: Internal transition
[10.0, 2:13:36.632] SimViewer_expframe.expframe.expframe.Upstream: Internal transition from Case1
[10.0, 2:13:36.632] SimViewer_expframe.expframe.expframe.Upstream: Holding in phase Case1_1 for time 1.0
[10.0, 2:13:36.632] Upstream - internal event for Case1  - time Elapsed: NA getTimeAdvance: 1.0 clock: 10.0
[10.0, 2:13:36.632] Upstream - internal event for Case1 - time Elapsed: NA getTimeAdvance: 10.0 clock: 10.0
[10.0, 2:13:36.632] SimViewer_expframe: Simulation step finished
[11.0, 2:13:36.773] SimViewer_expframe: Simulation step started
[11.0, 2:13:36.788] Upstream - output event for Case1_1 - time Elapsed: NA getTimeAdvance: 1.0 clock: 11.0
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.Upstream: Internal transition
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.Upstream: Internal transition from Case1_1
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.Upstream: Holding in phase Wait1 for time 49.0
[11.0, 2:13:36.788] Upstream - internal event for Case1_1 - time Elapsed: NA getTimeAdvance: 49.0 clock: 11.0
[11.0, 2:13:36.788] Upstream - internal event for Case1_1 - time Elapsed: NA getTimeAdvance: 1.0 clock: 11.0
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Received messages [inCtrlIn1: LaserMsg
        id: 1
        name: Case 1 Control Message 1
        status: 6
        magnitude: 0.0]
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: External transition
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Respond for time 0.0
[11.0, 2:13:36.788] @@************* CPG CONTROLLER - START PASSIVE CTRLIN1 EXTERNAL
EVENT********************************************************************
[11.0, 2:13:36.788] CPG Controller - external event for Passive CtrlIn with CtrlIn - time Elapsed: 11.0 getTimeAdvance: 0.0 clock: 11.0
[11.0, 2:13:36.788] CPG Controller received a laser fire message
[11.0, 2:13:36.788] CPG Controller - Preparing laser fire control message for: Case 1 Control Message 1 to port CtrlOut2
[11.0, 2:13:36.788] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Respond for time 0.0
[11.0, 2:13:36.788] @@************* CPG CONTROLLER - END PASSIVE CTRLIN1 EXTERNAL
EVENT********************************************************************
[11.0, 2:13:36.944] SimViewer_expframe: Simulation step finished
[11.0, 2:13:37.54] SimViewer_expframe: Simulation step started
[11.0, 2:13:37.54] @@************* CPG CONTROLLER - START RESPOND OUTPUT
EVENT********************************************************************
[11.0, 2:13:37.54] CPG Controller - output event for Respond - time Elapsed: NA getTimeAdvance: 0.0 clock: 11.0
[11.0, 2:13:37.54] sendCtrl getStatus =: 6
[11.0, 2:13:37.54] Last Classical Detection Optical Power =: 0.0
[11.0, 2:13:37.54] ### CPG Controller - Sending laser fire message: CPG Controller FIRE Response Message for Case 1 Control Message 1 to
port CtrlOut2
[11.0, 2:13:37.54] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 1 Environmental Received: 0 Control Received: 1 Control Received: 1
```

```
[11.0, 2:13:37.54] @@************* CPG CONTROLLER - END RESPOND OUTPUT
EVENT**********************************************************************
[11.0, 2:13:37.54] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Internal transition
[11.0, 2:13:37.54] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Internal transition from Respond
[11.0, 2:13:37.54] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Passive for time Infinity
[11.0, 2:13:37.54] @@************* CPG CONTROLLER - START RESPOND INTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.69] CPG Controller - internal event for Respond - time Elapsed: NA getTimeAdvance: Infinity clock: 11.0
[11.0, 2:13:37.69] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 1 Environmental Received: 0 Control Received: 1 Control Received: 1
[11.0, 2:13:37.69] CPG Controller - Ending Respond internal transition
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.cpgcontroller: Holding in phase Passive for time Infinity
[11.0, 2:13:37.69] @@************* CPG CONTROLLER - END RESPOND INTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: Received messages [inCtrlIn: LaserMsg
          id: 1
          name: CPG Controller FIRE Response Message for Case 1 Control Message 1
          status: 6
          magnitude: 0.0]
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: External transition
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase UpdateLaser for time 0.0
[11.0, 2:13:37.69] @@************* LASER - START PASSIVE CTRLIN EXTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.69] Laser - external event for Passive CtrlIn with CtrlIn - time Elapsed: 11.0 getTimeAdvance: 0.0 clock: 11.0
[11.0, 2:13:37.69] Laser is set to ON
[11.0, 2:13:37.69] Laser - Preparing laser fire control message for: CPG Controller FIRE Response Message for Case 1 Control Message 1 to port
CtrlOut
[11.0, 2:13:37.69] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase UpdateLaser for time 0.0
[11.0, 2:13:37.69] @@************* LASER - END PASSIVE CTRLIN EXTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.69] SimViewer_expframe: Simulation step finished
[11.0, 2:13:37.288] SimViewer_expframe: Simulation step started
[11.0, 2:13:37.288] @@************* LASER - START UPDATELASER OUTPUT
EVENT**********************************************************************
[11.0, 2:13:37.288] Laser - output event for UpdateLaser - time Elapsed: NA getTimeAdvance: 0.0 clock: 11.0
[11.0, 2:13:37.288] ******************* DISPLAY PULSES IN QUEUE *******************
[11.0, 2:13:37.288] Total Number of Optical Pulses in Queue  : 0 before reflection.
[11.0, 2:13:37.288] ***************************************************************
[11.0, 2:13:37.288] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 0 Environmental Received: 0 Control Received: 1
[11.0, 2:13:37.288] @@************* LASER - END UPDATELASER OUTPUT
EVENT**********************************************************************
[11.0, 2:13:37.303] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition
[11.0, 2:13:37.319] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition from UpdateLaser
[11.0, 2:13:37.319] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase Passive for time Infinity
[11.0, 2:13:37.319] @@************* LASER - START UPDATELASER INTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.319] Laser - internal event for UpdateLaser - time Elapsed: NA getTimeAdvance: Infinity clock: 11.0
[11.0, 2:13:37.319] ******************* DISPLAY PULSES IN QUEUE *******************
[11.0, 2:13:37.319] Total Number of Optical Pulses in Queue  : 0 after reflection.
[11.0, 2:13:37.319] ***************************************************************
[11.0, 2:13:37.319] InterruptRespond = false
[11.0, 2:13:37.319] needRespond = false
[11.0, 2:13:37.319] laserpower = true
[11.0, 2:13:37.319] currentStatus = 6
[11.0, 2:13:37.319] sendCtrl status is: 6
[11.0, 2:13:37.319] Laser - Preparing optical pulse for: Laser Output Pulse 1 to port OptOut1
[11.0, 2:13:37.319] Going to Create Pulse Phase Next!!!
[11.0, 2:13:37.319] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase CreatePulse for time 5.0
[11.0, 2:13:37.319] @@************* LASER - END UPDATELASER INTERNAL
EVENT**********************************************************************
[11.0, 2:13:37.319] SimViewer_expframe: Simulation step finished
[16.0, 2:13:37.459] SimViewer_expframe: Simulation step started
[16.0, 2:13:37.459] @@************* LASER - START CREATEPULSE OUTPUT
EVENT**********************************************************************
[16.0, 2:13:37.459] Laser - output event for CreatePulse - time Elapsed: NA getTimeAdvance: 5.0 clock: 16.0
[16.0, 2:13:37.459] ******************* DISPLAY PULSES IN QUEUE *******************
[16.0, 2:13:37.459] Total Number of Optical Pulses in Queue  : 0 BEFORE propagation.
[16.0, 2:13:37.459] ***************************************************************
[16.0, 2:13:37.459] ** OUTPUT EVENT FOR CREATEPULSE
[16.0, 2:13:37.459] ### Laser - Sending generated optical pulse: Laser Output Pulse 1 to port OptOut1
[16.0, 2:13:37.459] Pulses Received: 0 Reflected: 0 Queued: 0 Sent: 1 Environmental Received: 0 Control Received: 1
```

```
[16.0, 2:13:37.459] @@************* LASER - END CREATEPULSE OUTPUT
EVENT********************************************************************
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Internal transition from CreatePulse
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase Passive for time Infinity
[16.0, 2:13:37.459] @@************* LASER - START CREATEPULSE INTERNAL
EVENT********************************************************************
[16.0, 2:13:37.459] Laser - internal event for CreatePulse - time Elapsed: NA getTimeAdvance: Infinity clock: 16.0
[16.0, 2:13:37.459] ****************** DISPLAY PULSES IN QUEUE ******************
[16.0, 2:13:37.459] Total Number of Optical Pulses in Queue  : 0 BEFORE propagation.
[16.0, 2:13:37.459] ****************************************************************
[16.0, 2:13:37.459] *************** Adjust queue ***************
[16.0, 2:13:37.459] Total Number of Optical Pulses in Queue  : 0
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.laser: Holding in phase Passive for time Infinity
[16.0, 2:13:37.459] @@************* LASER - END CREATEPULSE INTERNAL
EVENT********************************************************************
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.pmfiber1: Received messages [inOptIn1: OpticalPulse
          id: 1
          name: Laser Output Pulse 1
          duration: 4.0E-10
          opticalPower: 8.709666395E-5
          brightPulseFlg: false
          amplitude: 1.94095418E-6
          centralFrequency: 1.0E15
          globalPhase: 0.0
          ellipticity: 0.0
          orientation: 0.0
          numberOfGaussians: 3
          gA0: 45.5
          gA1: 38.064
          gA2: 4.75752
          gM0: 9.54844E-11
          gM1: 1.81125E-10
          gM2: 3.52322E-10
          gSD0: 1.98151E-11
          gSD1: 5.81389E-11
          gSD2: 4.90169E-11]
[16.0, 2:13:37.459] SimViewer_expframe.expframe.expframe.cpg.pmfiber1: External transition
```

B.4 References

Chow, A. C. (1996). Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation, 13*(2), 55-68. Retrieved from http://www.bgc-jena.mpg.de/~twutz/devsbridge/pub/chow96_parallelDEVS.pdf

Sargent, R. G. (2005). Verification and validation of simulation models. Paper presented at the *Proceedings of the 37th Conference on Winter Simulation,* 130-143. Retrieved from http://student.telum.ru/images/6/66/Sargent_VV_2010.pdf

Zeigler, B. P. (1976). *Theory of modeling and simulation*. New York: John Wiley & Sons, Inc.

Zeigler, B. P. (1984). *Multifaceted modelling and discrete event simulation*. London, UK: Academic Press.

Appendix C – Cryptography Overview

The need for secure communications has existed since the dawn of humanity. Cryptography, the practice and study of techniques for securing communications between two authorized parties in the presence of one or more unauthorized third parties, is an essential tool used to assure information security (Piper & Murphy, 2002). Historically, government and military applications are the chief users of cryptography, but today cryptography provides security services to almost everyone, including confidentiality, integrity, authentication, authorization, and non-repudiation (Barker et al., 2005).

C.1 Cryptosystems

A cryptosystem is composed of two basic components: an algorithm and one or more keys. The algorithm is the mathematical transformation used to encrypt and decrypt messages and the key(s) are parameters used in the encryption and decryption processes. Figure 1 shows a block diagram of a simple cryptosystem. The original message, m, called the "plaintext" transforms into the "ciphertext", $EK(m)$, using the encryption algorithm, E, and the encryption key, K. The terms plaintext and ciphertext refer to binary data and can represent anything in digital form (e.g., text, audio, video, pictures, and programs). Other parameters (e.g., initialization vectors, salt, etc.) may be used but are not shown for simplicity. Ideally, the ciphertext is not decipherable unless you possess the matching decryption key. The transformation of plaintext into ciphertext "protects" the confidentiality of messages transmitted over a public channel where an adversary could possibly intercept it. Upon receipt, the ciphertext message is transformed back into the plaintext, m, using the decryption algorithm, D, and the decryption key K'. The decryption algorithm, D, is the inverse transformation of the encryption algorithm, *M*, which means that *DK′* (*EK*(*m*))=*m*. Note that in general *K* does not have to equal *K′*, although it does for symmetric algorithms.

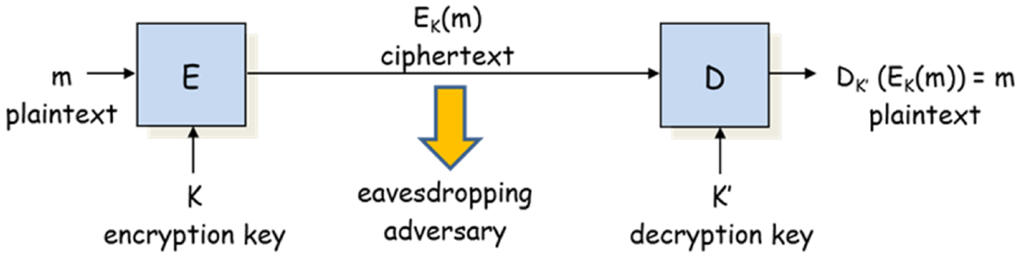


Figure 21. A Simple Cryptosystem Block Diagram

There are three basics types of cryptographic algorithms: symmetric, asymmetric and hashing functions. Symmetric key algorithms use the same key (e.g., *K* = *K′*) for encryption and decryption. The benefits of a symmetric key algorithm are that it provides confidentiality, is fast, is easily implemented in hardware, and consumes little computational power when compared to asymmetric algorithms. However, symmetric key algorithms only provide confidentiality and require a separate key for each pair of entities who wish secure communications, which does not scale well when large numbers of entities must securely communicate. Examples of symmetric key algorithms include the Data Encryption Standard (DES), 3-DES, AES, Blowfish, Rivest Cipher 4 (RC4), and Rivest Cipher 5 (RC5) (Loepp & Wootters, 2006; Piper & Murphy, 2002; Schneier, 1995).

Asymmetric key algorithms used mathematically related, but different (e.g., *K* ≠ *K′*), key pairs for encryption and decryption (e.g., public and private keys) which reduces the key distribution burden. The benefits of an asymmetric key algorithm are: 1) no need for “out of band” (sending the key through a different channel than the encryption channel) key distribution as public keys are freely shared, 2) it scales better since each individual only a needs a single key-pair, 3) provides authentication and non-repudiation. However, asymmetric key algorithms are slow because they require complex mathematical operations and typically consume more computational power than symmetric algorithms. Examples of asymmetric algorithms include the Rivest Shamir Adleman (RSA), Pretty Good Privacy (PGP), El Gamal, Elliptic Curve Cryptography (ECC), and Diffie-Hellman (Loepp & Wootters, 2006; Piper & Murphy, 2002; Schneier, 1995).

Hash algorithms are one-way functions that take an arbitrary-length message and create a fixed-length message digest. Hash algorithms may or may not require a key depending on their mode of operation. Hashing provides an efficient way to check the integrity of stored or transmitted data without having to compare the data bit by bit. However, since hash algorithms map all possible inputs to a fixed length message digest, more than one input can map to the same digest creating a "collision" which may provide an advantage to an eavesdropper. Examples of hash algorithms include Message Digest-4 (MD-4), Message Digest-5 (MD-5), and Secure Hash Algorithm-1 (SHA-1) (Loepp & Wootters, 2006; Piper & Murphy, 2002; Schneier, 1995).

C.2 The One-Time Pad (OTP)

The only cryptographic algorithm mathematically proven as unconditionally secure is the OTP. The OTP is a symmetric cryptographic algorithm and is relatively easy to understand. The first known description of the OTP was in 1882 when Frank Miller described "superencipherment" as a means to insure the privacy and secrecy of telegraphic communications (Bellovin, 2011). Miller's method required the use of a randomly created key that was never reused. In 1917, Gilbert Vernam invented, and later patented, a cipher based on teleprinter technology, but it was vulnerable as it reused key material (Vernam, 1919; Vernam, 1926). Despite this weakness, a National Security Agency (NSA) report identified Vernam's patent as "perhaps one of the most important in the history of cryptography" (Kahn, 1996). In the 1940s, Claude E. Shannon proved the theoretical significance of the security of the OTP (C. E. Shannon, 1949).

The strength most modern cryptographic algorithms relies on "computational security," meaning the algorithm is considered secure if there is a negligible probability of determining the key in a "reasonable" amount of time using current technology (Schneier, 1995). In theory, every cryptographic algorithm, except the OTP, is breakable if an adversary has enough captured ciphertext, computational resources, and time. However, the OTP is not commonly used due to the large amount of non-reusable key material required to properly use the algorithm. To take advantage of the OTP, the sender (historically known as Alice) creates and distributes random secret keys to the receiver (historically known as Bob) equal in length to all messages exchanged. In practice, this places a significant burden on key distribution and management as one must continually generate and securely distribute key pads between the senders and receivers (Schneier, 1995). Historically, the OTP is only used in environments which justify the costs involved with secure key distribution (Singh, 1999).

C.3 Computational Security

Along with problems in key distribution, there are concerns that certain cryptographic algorithms will become useless when quantum computers with large number of quantum bits (qubits) become available. Each qubit allows a quantum computer to perform in a single step what takes multiple steps in existing computers (Nielsen & Chuang, 2010). This allows a quantum computer to complete difficult tasks much sooner than a regular computer, such as the factoring of large numbers (Shor, 1997). Unfortunately, most current encryption algorithms rely on the factoring of large numbers being a hard or insurmountable problem for computers, so quantum computing would risk the security of these schemes (Institute for Quantum Computing, 2013)

C.4 References

Barker, E. B., Barker, W. C., & Lee, A. (2005). *NIST special publication 800-21 guideline for implementing cryptography in the federal government* NIST. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-21-1/sp800-21-1_Dec2005.pdf

Bellovin, S. M. (2011). Frank miller: Inventor of the one-time pad. *Cryptologia, 35*(3), 203-222. Retrieved from http://academiccommons.columbia.edu/download/fedora_content/download/ac:135404/CONTENT/cucs-009-11.pdf

Institute for Quantum Computing. (2013). Quantum computing 101. Retrieved, 2013, Retrieved from http://iqc.uwaterloo.ca/welcome/quantum-computing-101

Kahn, D. (1996). *The codebreakers: The comprehensive history of secret communication from ancient times to the internet* (2nd ed.). New York: Scribner.

Loepp, S., & Wootters, W. K. (2006). *Protecting information: From classical error correction to quantum cryptography* (1st ed.). New York: Cambridge University Press.

Nielsen, M. A., & Chuang, I. L. (2010). *Quantum computation and quantum information* (10th ed.). Cambridge, UK: Cambridge university press.

Piper, F., & Murphy, S. (2002). *Cryptography A very short introduction* (5th ed.). New York: Oxford University Press.

Schneier, B. (1995). In Sutherland P. (Ed.), *Applied cryptography: Protocols, algorithms, and source code in C* (2nd ed.). New York: John Wiley & Sons, Inc.

Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell System Technical Journal, 28*(4), 656-715. Retrieved from http://dm.ing.unibs.it/giuzzi/corsi/Support/papers-cryptography/Communication_Theory_of_Secrecy_Systems.pdf

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing, 26*(5), 1484-1509.

Singh, S. (1999). *The code book: The secret history of codes and code-breaking* (17th ed.). London: Fourth Estate.

Vernam, G. S. (1919). *Secret signaling system*. New York: Retrieved from http://www.google.com/patents?hl=en&lr=&vid=USPAT1310719&id=1BpPAAAAEBAJ&oi=fnd&dq=vernam+1919&printsec=abstract

Vernam, G. S. (1926). Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Transactions of the American Institute of Electrical Engineers, 45*, 295-301. Retrieved from http://math.boisestate.edu/~liljanab/MATH509Spring2012/vernam.pdf

Appendix D - Bandpass Filter

D.1 Device Description:

The Bandpass filter is a device that allows light around a central frequency to pass through in either direction, along with a small band of frequencies on either side of the central frequency. The filter creates a Fabry-Perot etalon (Saleh & Teich, 1991) (cavity) to create destructive interference to block light outside of the "pass through" area. The passband region is "tent" shaped with small bands on either side. See Figure 1 for an example of this "tent" structure.

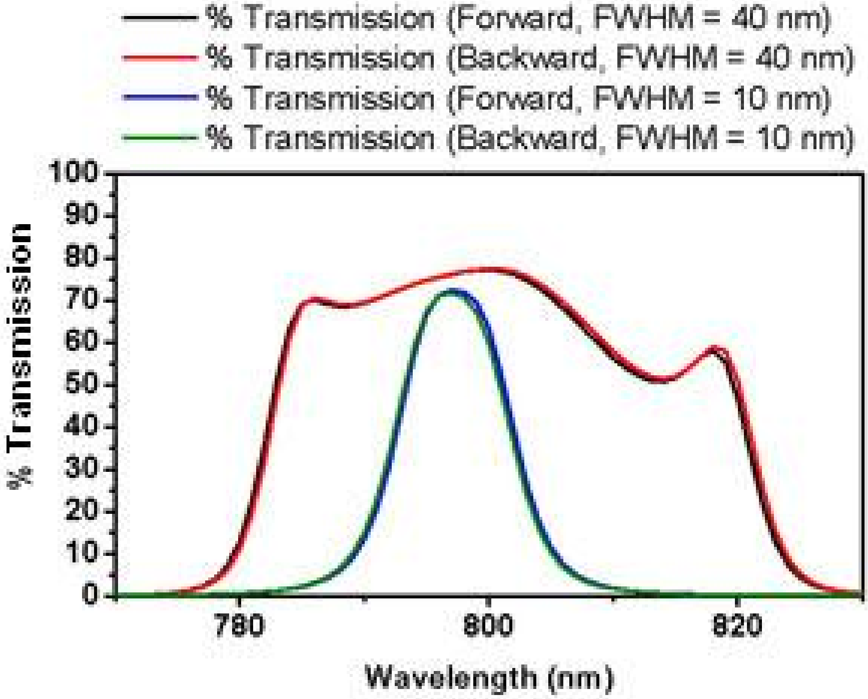


Figure 22. FB800-10 and FB800-40 filter passbands (ThorLabs, 2013).

The Bandpass filter is a series of substrates with material designed to block light of certain wavelengths. Between the substrates is dielectric material with specific thickness based on the wavelength of the passthrough along with spacer layers. The Fabry-Perot cavity is formed by the layering of the spacers and the dielectric material. The sandwiched material is then mounted in a protective chassis. See Figure2.

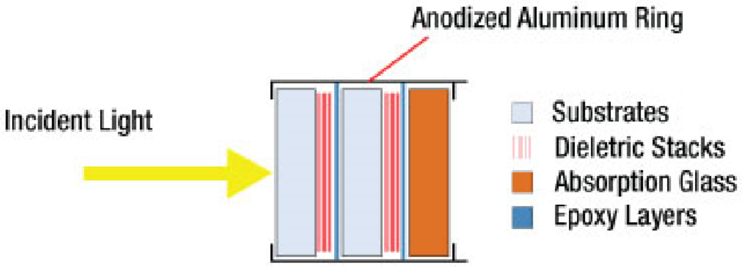


Figure 23. Schematic of a typical bandpass filter (ThorLabs, 2013).

The Bandpass filter is a bidirectional optical component with two optical ports. Optical signals arriving at the input port are propagated to the other port after a defined propagation delay and the filter is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the Bandpass filter may become permanently damaged which changes its propagation characteristics. Similarly, the Bandpass filter is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the Bandpass filter may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the Bandpass filter is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the Bandpass filter. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the Bandpass filter to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the Bandpass filter using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the Bandpass filter. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

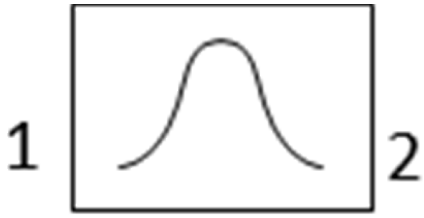


Figure 24. Symbol for the Bandpass filter in the QKD system architecture.

D.2 Bandpass filter Conceptual Model

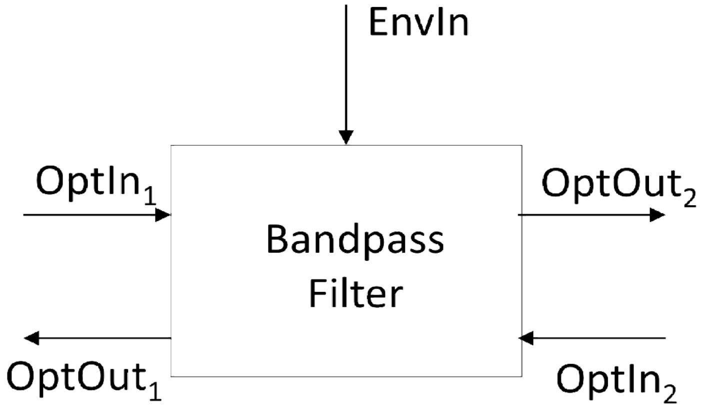


Figure 25. Bandpass filter conceptual model.

The conceptual model for a Bandpass filter consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the Bandpass filter. In comparison to the Bandpass filter symbol used in the QKD simulation architecture shown in Figure 3, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the Bandpass filter, a small portion of the signal will be instantaneously reflected back towards the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn$_1$ in Fig. 4 will instantaneously generate a reflected emitting out of OptOut$_1$.

The Bandpass filter calculates the power of the incoming packet through either optical port after a time equaling the propagation delay of the module at full power minus some small amount to account for attenuation through the device. Even though the bandpass filter is meant to block light that does not match the central frequency, a small amount of the light on either side of the central frequency will make it through the device and out the input port.

The Bandpass filter must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the Bandpass filter can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

D.3 Mathematical Model

For a detailed mathematical description of the Bandpass filter, refer to Section 2.8 which contains the Mathematica worksheet provided by the optical physics SME.

D.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the Bandpass filter.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Remove the packet from the queue, calculate the attenuated output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

D.5 Phase Transition Diagram

The phase transition diagram in Fig. 5 shows the phases of the Bandpass filter in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the Bandpass filter at the same time.

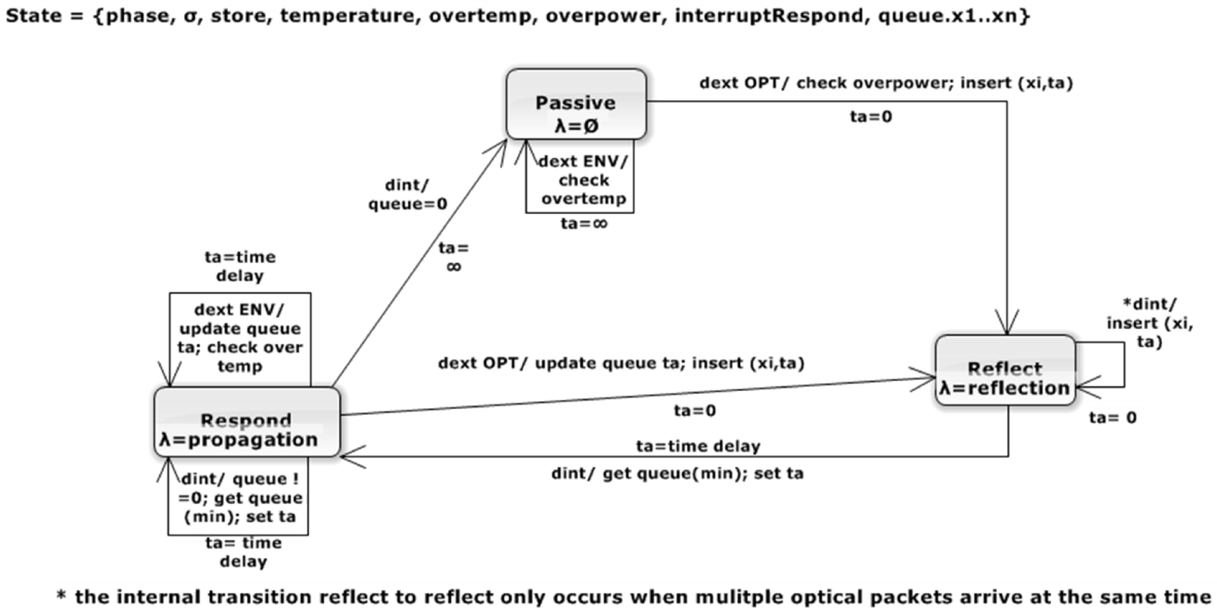

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
Respond
λ=propagation
ta=0
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue != 0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 26. Bandpass filter phase transition diagram.

D.6 Event-Trace Diagram

This section shows various examples of packets entering the Bandpass filter. The tables list the states the bandpass filter proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the Bandpass filter, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable

- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

D.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 10. *Case I state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	
5	s3	entry	passive	inf	x1	c	n	n	n	null	

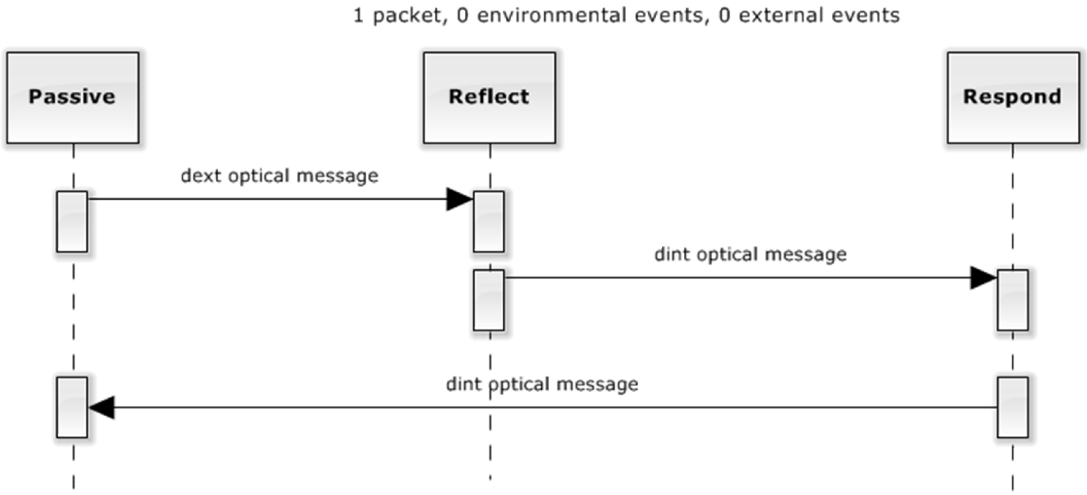


Figure 27. Case I sequence diagram.

D.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 11. *Case II state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	Interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5

	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

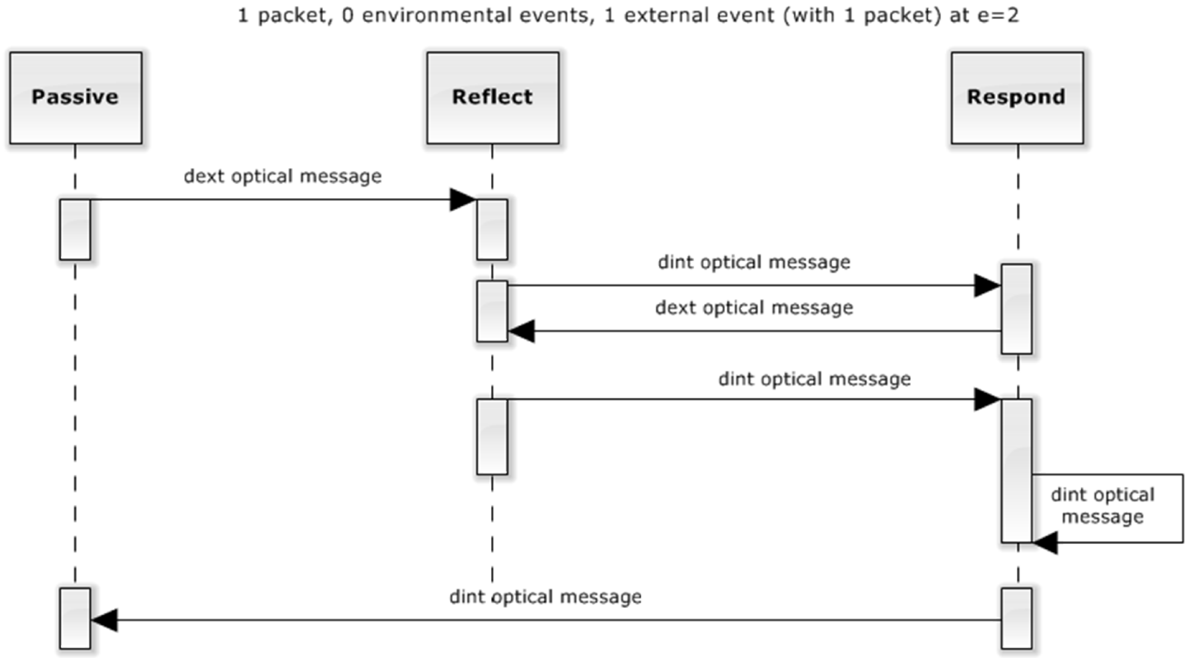


Figure 28. Case II sequence diagram.

D.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 12. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

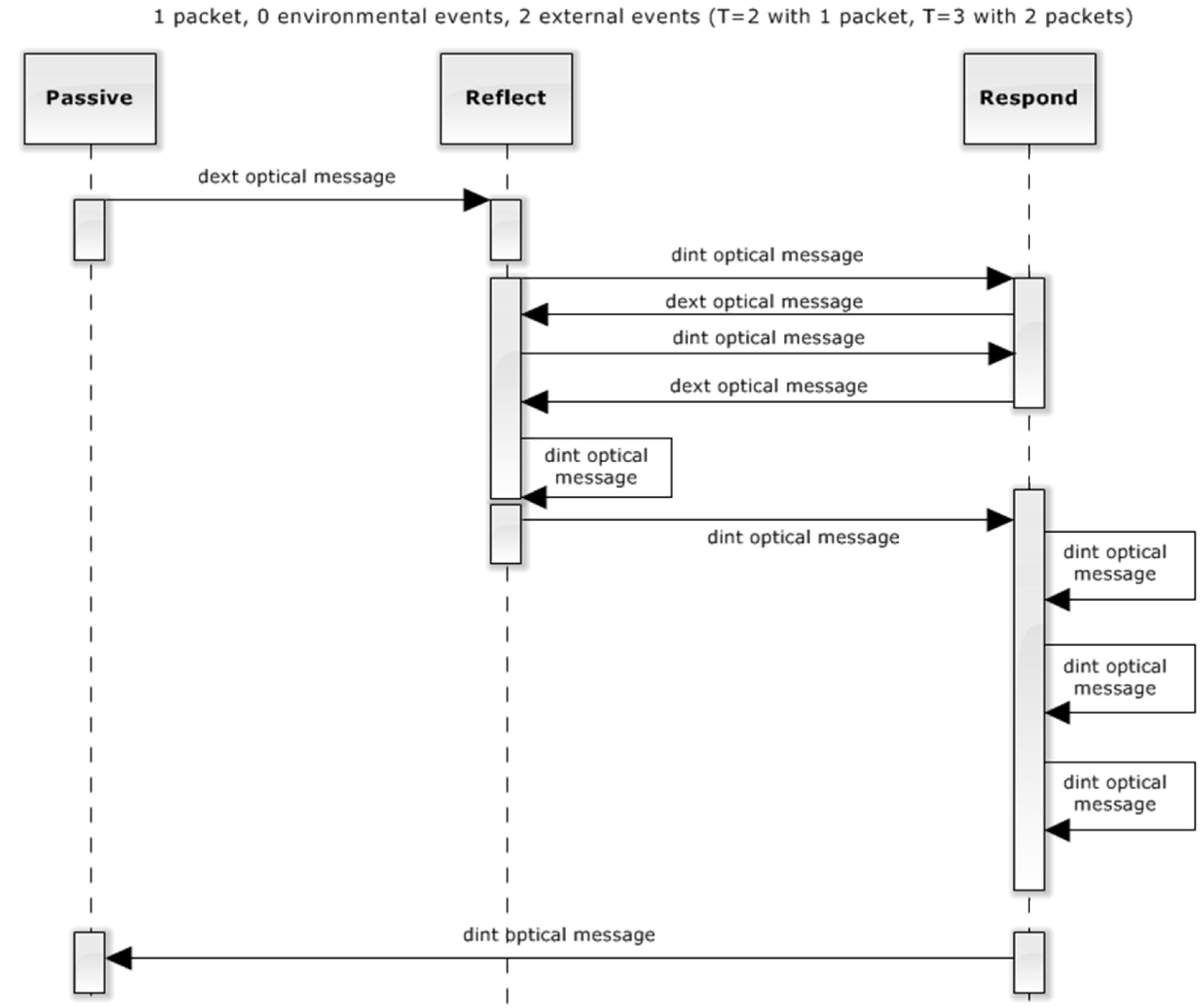


Figure 29. Case III sequence diagram.

D.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 13. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

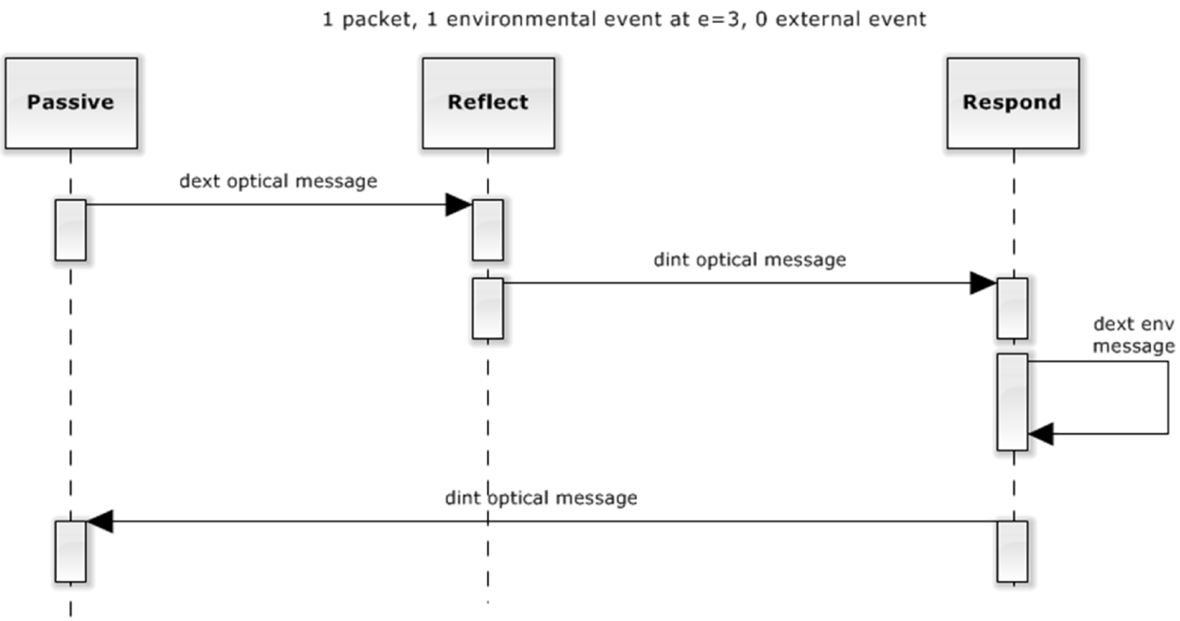


Figure 30. Case IV sequence diagram.

D.7 Bandpass filter Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.

- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the Bandpass filter we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{Bandpass\ filter}$* $\boldsymbol{= (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator

phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “reflect”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time $delay_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
$q.v_{min}$ = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“OptIn$_1$”, “OptIn$_2$”, “EnvIn”} with
X_M = {(“OptIn$_1$”, V_{opt}), (“OptIn$_2$”, V_{opt}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“OptOut$_1$”, “OptOut$_2$”} with
Y_M = {(“OptOut$_1$”, $Y_{OptOut1}$), (“OptOut$_2$”, $Y_{OptOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond, queue*} = {{“passive”, “reflect”, “respond”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x *V*}

External Transition Function:

δ_{ext}(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =

```
(“reflect”, 0, store, temperature, overtemp, overpower,interruptRespond, queue.x1..xn)
 if phase = “passive” and p ∈ {“OptIn1”, “OptIn2”}
    for messagebag != null
       current = messagebag_first()
        if current.value.power > damaged.power
           overpower = “Y”
        insert_event_q(current)
        remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = “respond” and p ∈ {“OptIn1”, “OptIn2”}
    update_delay(queue)
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
```

```
        overpower =  "Y"
      insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_need_reflected()
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
   interruptRespond= "Y"
   timeLeftRespond = timeLeftRespond - e

 ("passive", ∞, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = "passive" and p = "EnvIn"
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = "Y"

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                   queue.x1..xn)
   if phase = "respond" and p = "EnvIn"
     update_delay(queue)
     timeLeftRespond = time_delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"
     time.delay = timeLeftRespond
```

Internal Transition Function:

```
$\delta_{int}$(phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue) =
("reflect", 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
  if phase = "reflect" and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)

 ("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
  if phase = "reflect" and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = "N"
      current = queue_min()
      time.delay = current.time.delay
      if InPort = "OptIn$_1$"
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "OptOut$_2$"
      if InPort = "OptIn$_2$"
```

```
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_1$”
      timeLeftRespond = propagation delay
    else
      time.delay = timeLeftRespond

  (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if InPort = “OptIn$_1$”
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_2$”
      if InPort = “OptIn$_2$”
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_1$”
      interruptRespond= “N”

  (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and  size  = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (output.port, output.pulse)
    if phase = “respond”

  Ø (null output)
    otherwise;
```

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower, interruptRespond, queue) = \sigma$;

D.8 Mathematical model

Pulse propagation considerations for the Bandpass Filter Module within the QKD OMNet++ simulation environment

There a several device designs which can achieve bandpass filtering in an all-fiber or fiber-based platform. Chirped fiber gratings, tuned acousto-optic fiber, and integrated micro-optic devices are all examples fiber-based bandpass filters. More research needs to be conducted on these devices to form a physically accurate and complete model. In the interest of expediting the construction of the simulation enviroment, we consider here the basic functions of a fiber-based bandpass filter. The bandpass wavelength region will be considered to be a "tent-shaped", with roll-offs on either side of the central wavelength, finally leveling at the maximum rejection attenuation.

The operational characteristics are as follows:

- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

The only significant modification to the optical message will be the amplitude (power), dependent upon the wavelength.

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\,\omega_o\,t}\,e^{i\,\theta} \begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Bandpass Filter Module

```
Ein := Eo (* electric field input into port 1 *)
ωo := 2 π * (2.99792458 * 10^8) / (1.54825 * 10^-6)
   (* angular frequency of optical wavelength, units of Rad/s *)
```

The following parameter values are used as an example, and are taken from a tunable bandpass fiber optic filter offered by newport optics (http://www.newport.com/Tunable-Bandpass-Fiber-Optic-Filter/835502/1033/info.aspx#tab_Overview).

```
InsertionLoss := 1.5 (* insetion power loss at the bandpass wavelength, units of -dB *)
BandPassWavelength := 1550 * 10^-9 (* central wavelength of bandpass window, units of m *)

MidBandLoss := 0.5 (* loss in addition to InBandLoss,
at MidBandWidth (+/-nm) from BandPassWavelength (nm), units of -dB *)
MidBandWidth := 1.0 * 10^-9 (* total width of MidBandLoss window, units of m *)

OutBandLoss := 20
(* power loss outside of the bandpass wavelength window, units of -dB *)
OutBandWidth := 3.5 * 10^-9 (* total width of bandpass window, units of m *)

RetLoss := 50 (* typical return loss,
signal reflected by an input beam, units of -dB *)
TempH := 65 (* max operational temperature, units of °C *)
TempL := 15 (* min operational temperature, units of °C *)
MaxPwr := 500 (* maximum operational power, units of mW *)
```

Attenuation Calculations for Bandpass Filter

$$\lambda\text{in} := 2\,\pi * \left(2.99792458 * 10^{8}\right) / \omega\text{o} \quad \text{(* input wavelength calculated from given } \omega\text{o *)}$$

$$\text{MidBandLow} := \text{BandPassWavelength} - \frac{\text{MidBandWidth}}{2}$$

$$\text{MidBandHigh} := \text{BandPassWavelength} + \frac{\text{MidBandWidth}}{2}$$

$$\text{OutBandLow} := \text{BandPassWavelength} - \frac{\text{OutBandWidth}}{2}$$

$$\text{OutBandHigh} := \text{BandPassWavelength} + \frac{\text{OutBandWidth}}{2}$$

Psuedo Code; if λin = BandPassWavelength,

$$\text{Eout2[Ein\_, InsertionLoss\_]} = \text{Ein} * \sqrt{10^{-\text{InsertionLoss}/10}}$$

Psuedo Code; else if λin > BandPassWavelength && λin ≤ MidBandHigh,

$$\text{Eout2[Ein\_, InsertionLoss\_, }\lambda\text{in\_, BandPassWavelength\_, MidBandWidth\_, MidBandLoss\_]} =$$

$$\text{Ein} * \sqrt{10^{-\text{InsertionLoss}/10}} * \sqrt{10^{-\left(\frac{\lambda\text{in}-\text{BandPassWavelength}}{(\text{MidBandWidth}/2)}\right)*\text{MidBandLoss}/10}}$$

0.794328 Eo

Psuedo Code; else if λin < BandPassWavelength && λin ≥ MidBandLow,

$$\text{Eout2[Ein\_, InsertionLoss\_, }\lambda\text{in\_, BandPassWavelength\_, MidBandWidth\_, MidBandLoss\_]} =$$

$$\text{Ein} * \sqrt{10^{-\text{InsertionLoss}/10}} * \sqrt{10^{-\left(\frac{\text{BandPassWavelength}-\lambda\text{in}}{(\text{MidBandWidth}/2)}\right)*\text{MidBandLoss}/10}}$$

Psuedo Code; else if λin > MidBandHigh && λin ≤ OutBandHigh,

$$\text{Eout2[Ein\_, InsertionLoss\_, MidBandLoss\_, }\lambda\text{in\_,}$$
$$\text{MidBandHigh\_, MidBandLoss\_, OutBandHigh\_, OutBandLoss\_]} =$$

$$\text{Ein} * \sqrt{10^{-\text{InsertionLoss}/10}} * \sqrt{10^{-\text{MidBandLoss}/10}} * \sqrt{10^{-\left(\frac{\lambda\text{in}-\text{MidBandHigh}}{(\text{OutBandHigh}-\text{MidBandHigh})}\right)*(\text{OutBandLoss}-\text{MidBandLoss})/10}}$$

Psuedo Code; else if λin < MidBandLow && λin ≥ OutBandLow,

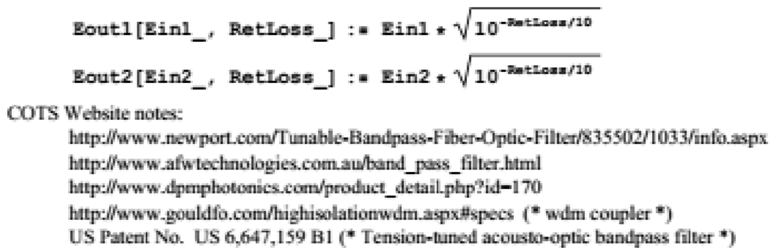
$$\text{Eout2[Ein\_, InsertionLoss\_, MidBandLoss\_, }\lambda\text{in\_,}$$
$$\text{MidBandHigh\_, MidBandLoss\_, OutBandHigh\_, OutBandLoss\_]} =$$

$$\text{Ein} * \sqrt{10^{-\text{InsertionLoss}/10}} * \sqrt{10^{-\text{MidBandLoss}/10}} * \sqrt{10^{-\left(\frac{\text{MidBandLow}-\lambda\text{in}}{(\text{MidBandLow} - \text{OutBandLow})}\right)*(\text{OutBandLoss}-\text{MidBandLoss})/10}}$$

Psuedo Code; else,

$$\text{Eout2[Ein\_, InsertionLoss\_, OutBandLoss\_]} = \text{Ein} * \sqrt{10^{-\text{InsertionLoss}/10}} * \sqrt{10^{-\text{OutBandLoss}/10}}$$

If we wish to flag the attenuator to include **undesired return (reflected)** messages, the following operations would hold true,



```
Eout1[Ein1_, RetLoss_] := Ein1 * √(10^(-RetLoss/10))

Eout2[Ein2_, RetLoss_] := Ein2 * √(10^(-RetLoss/10))
COTS Website notes:
        http://www.newport.com/Tunable-Bandpass-Fiber-Optic-Filter/835502/1033/info.aspx
        http://www.afwtechnologies.com.au/band_pass_filter.html
        http://www.dpmphotonics.com/product_detail.php?id=170
        http://www.gouldfo.com/highisolationwdm.aspx#specs  (* wdm coupler *)
        US Patent No.  US 6,647,159 B1 (* Tension-tuned acousto-optic bandpass filter *)
```

D.9 Component Use Cases

D.9.1 Respond to an Optical Packet in the Bandpass filter

Optical packet arrives at the bandpass filter. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the bandpass filter. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated optical output signal based on the input signal frequency, the bandpass central frequency and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
  - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state

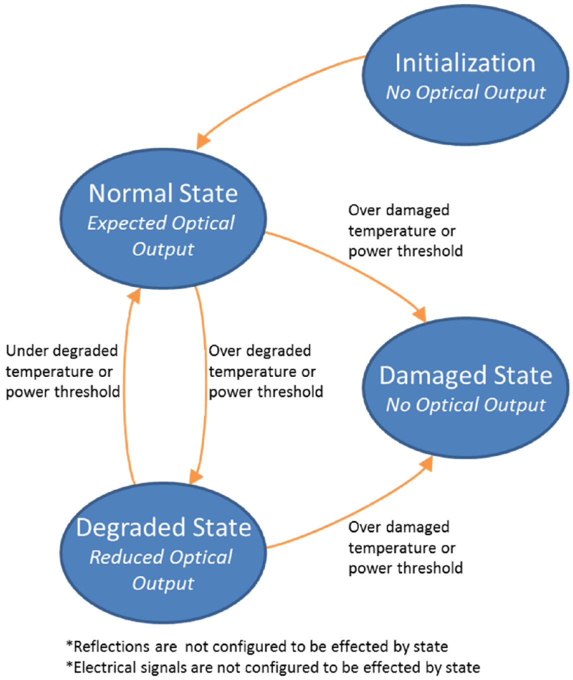

Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Damaged State
No Optical Output
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold
*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 31. Component states.

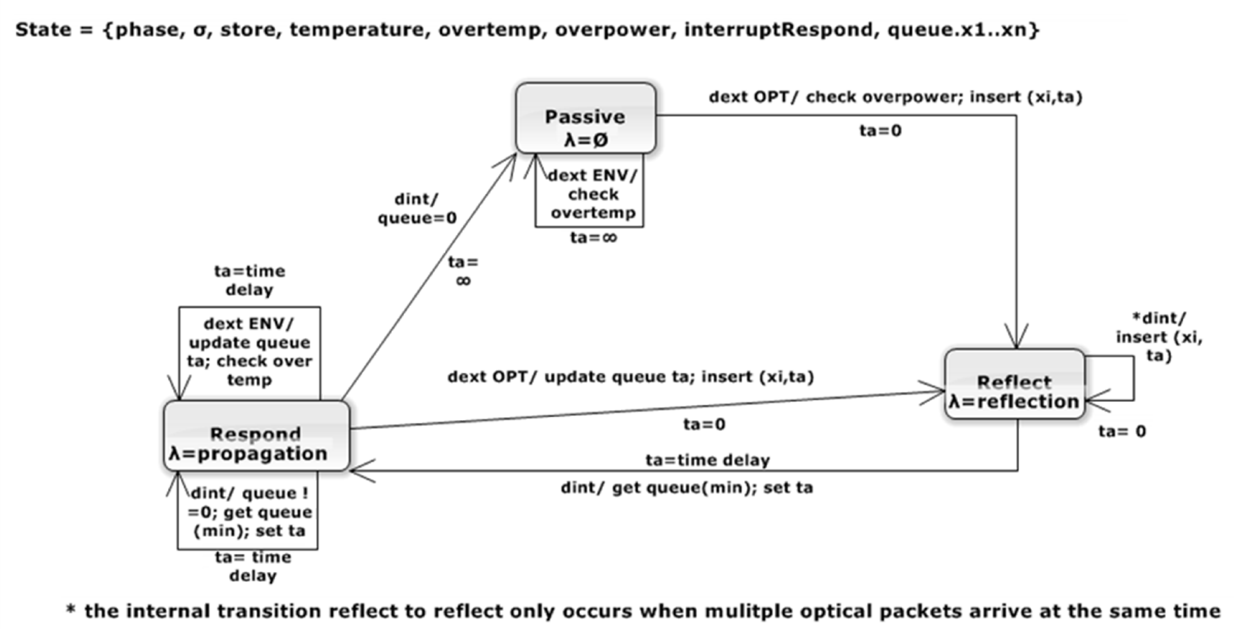

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
Respond
λ=propagation
ta=0
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue ! =0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 32. Bandpass phase transition diagram.

D.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.

- Optical packet attenuated properly to the limit of accuracy.
- Optical packet propagated out the correct port at the correct time.

D.9.3 Respond to an Environmental Packet in the Bandpass filter

Environmental packet arrives at the bandpass filter. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

D.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

D.10 Component Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Bandpass Test Cases.*

		Inject Port				Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Env**	Note	opt #	**env #**
Passive	1	1	0	0	single	1	0
	2	0	1	0	single	2	0
	3	0	0	1	single	2	1
	4	1	1	0	same time	4	1
	5	1	1	0	differ time	6	1
	6	1	1	1	same time	8	2
	7	1	1	1	differ time	10	3
	8	0	1	1	same time	11	4
	9	0	1	1	differ time	12	5
	10	1	0	1	same time	13	6
	11	1	0	1	differ time	14	7
	20	2	0	0	same time	16	7
	21	0	2	0	same time	18	7
	22	2	2	0	same time	22	7
	23	2	2	0	differ time	26	7
	24	2	2	1	same time	30	8
	25	2	2	1	differ time	34	9

	26	0	2	1	same time	36	10
	27	0	2	1	differ time	38	11
	28	2	0	1	same time	40	12
	29	2	0	1	differ time	42	13
totals		21	21	13	42		
Respond	41	2	0	0	single	44	13
	42	0	2	0	single	46	13
	43	1	0	1	single	47	14
	44	2	1	0	same time	50	14
	45	2	1	0	differ time	53	14
	46	2	1	1	same time	56	15
	47	2	1	1	differ time	59	16
	48	0	2	1	same time	61	17
	49	0	2	1	differ time	63	18
	50	2	0	1	same time	65	19
	51	2	0	1	differ time	67	20
	60	3	0	0	same time	70	20
	61	0	3	0	same time	73	20
	62	3	2	0	same time	78	20
	63	3	2	0	differ time	83	20
	64	3	2	1	same time	88	21
	65	3	2	1	differ time	93	22
	66	0	3	1	same time	96	23
	67	0	3	1	differ time	99	24
	68	3	0	1	same time	102	25
	69	3	0	1	differ time	105	26
totals		36	27	13	63		
	TC1	1	0	2	single	106	28
	TC2	1	0	2	single	107	30
	TC3	1	0	2	single	108	32
	TC4	1	0	2	single	109	34
totals		4	0	8	12		

D.11 References

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). Bandpass filter structure. Retrieved September 16, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1000

Appendix E - Beamsplitter

E.1 Device Description:

The beamsplitter is an optical device used to split a beam of light in to a reflected beam and a transmitted beam and can be used to combine two beams of light into one stream (Saleh & Teich, 1991). In practical terms, light from one input fiber is sent through a collimating lens, then divided by the beamsplitter optic and focused on the output ports. Different fibers can be used on each port and different types of beamsplitting material can be used inside the housing.

A beamsplitter can be made from housing with collimating lenses and some form of a beamsplitting material, such as a partially reflective mirror or a think glass plate, which splits the light (Saleh & Teich, 1991). Physical designs include cubes mounted into brackets for free-space optics and housings that have permanently mounted pigtails or connectors for fiber lines and pure fiber devices commonly known as couplers. The amount of light directed to each port is variable and determined by the material inside the beamsplitter. Many combinations of splitting ratios exist, some of the most common are 50:50, 90:10, 70:30, but the devices can be made with almost any ratio. See Figure 1 for an example of a four-port free-space beamsplitter.

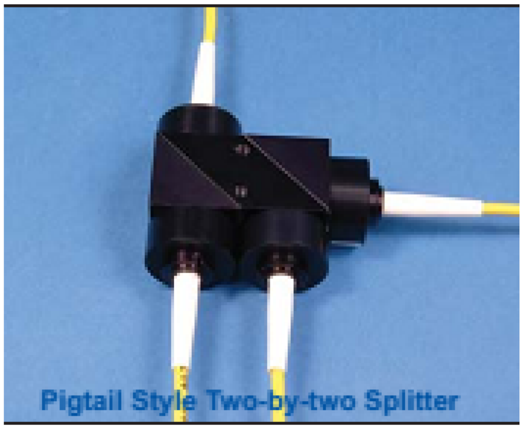


Figure 33. View of a free-space beamsplitter with fiber inputs (OZOptics, 2013).

Although beamsplitters may have from three to eight or more ports, this research will use the four port beamsplitting devices with fiber pigtails, per the discussion with the SME. In this research, the beamsplitter is a bidirectional optical component with four optical ports. Light entering the primary port splits into two beams and exits through the two output ports with the splitting ratio dependent on the device. In the opposite direction, the component works the same way, splitting the light by passing through a portion of the beam and reflecting the rest to the port opposite the reflected port used by the primary path.

The light suffers a slight attenuation from the material as it passes through the device and the splitting medium has a phase effect for the reflected portion of the beam. The reflected beam undergoes a global phase shift of $\pi/2$ and is applied to light passing through the device in both directions.

The internal material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the beamsplitter may become permanently damaged which changes its propagation characteristics. Similarly, the beamsplitter is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the beamsplitter may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the beamsplitter is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the beamsplitter. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the beamsplitter to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the beamsplitter using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the beamsplitter. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

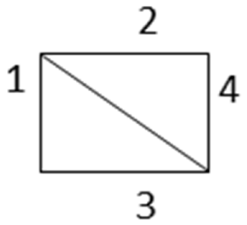


Figure 34. Symbol for the 4-port beamsplitter in the QKD system architecture.

E.2 Beamsplitter Conceptual Model

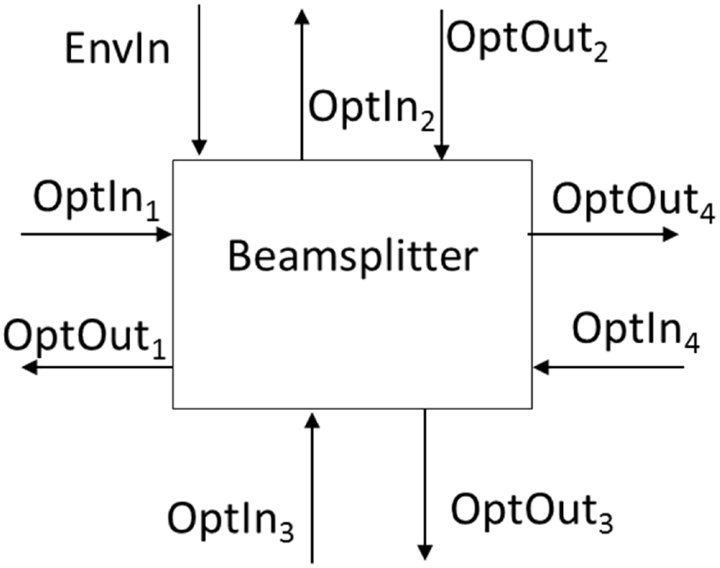


Figure 35. Beamsplitter conceptual model.

The conceptual model for a beamsplitter consists of four optical input ports {OptIn$_1$, OptIn$_2$, OptIn$_3$, OptIn$_{4\}}$, four optical output ports {OptOut$_1$, OptOut$_2$, OptOut$_3$, OptOut$_4$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the beamsplitter. In comparison to the beamsplitter symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the beamsplitter, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn$_1$ in Fig. 3 will instantaneously generate a reflected emitting out of OptOut$_1$.

The beamsplitter calculates changes to the power (attenuation) of any packet coming through an optical port after a time equaling the propagation delay of the module. The packet is calculated at full power minus some small amount to account for attenuation through the device. The model splits each incoming optical packet into a 'passed' packet and a 'reflected' packet, with the strength of each packet determined by the beamsplitting ratio, and injecting these packets into the queue. Each of these entries are a (port, value) pair, just as any other entry into the queue, with the [port] entry equal to the output port and the [value] equal to the adjusted values of the incoming packet. Additionally, packets output on the reflected port rotate $\pi/2$ (i.e. $\alpha = \alpha + \pi/2$) due to the effects of the beamsplitting material inside the device and is applied by the 'passed' and 'reflected' functions.

The beamsplitter calculates the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the beamsplitter can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

E.3 Mathematical Model

For a detailed mathematical description of the beamsplitter, refer to Section 3.8 which contains the Mathematica worksheet provided by the optical physics SME.

E.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the beamsplitter.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Determine the input port number.
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Remove the packet from the queue and split it into two packets
- Update the values for one packet as a ‘passed’ optical signal based on the characteristics of the beamsplitter, the original values of the input optical signal and the current environment and set the correct output port.
- Update the values for the other packet as a ‘reflected’ optical signal based on the characteristics of the beamsplitter, the original values of the input optical signal and the current environment and set the correct output port.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

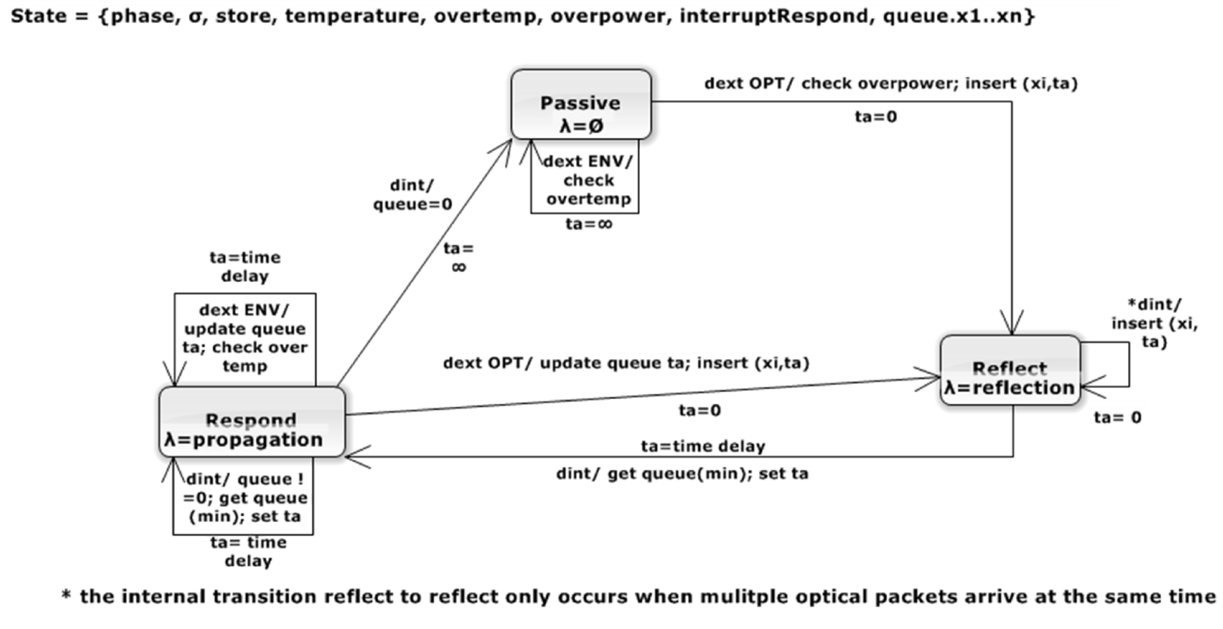
When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.

- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

E.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the beamsplitter in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the beamsplitter at the same time.



State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}

Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
dext OPT/ update queue ta; insert (xi,ta)
ta=0
*dint/ insert (xi, ta)
Reflect
λ=reflection
ta= 0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue !=0; get queue (min); set ta
ta= time delay

* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 36. beamsplitter phase transition diagram.

E.6 Event-Trace Diagram

This section shows various examples of packets entering the beamsplitter. The tables list the states the beamsplitter proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the beamsplitter, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

E.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 14. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (x*i*, *tp*)**	**Notes: assume tp=5**
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	
5	s3	entry	passive	inf	x1	c	n	n	n	null	

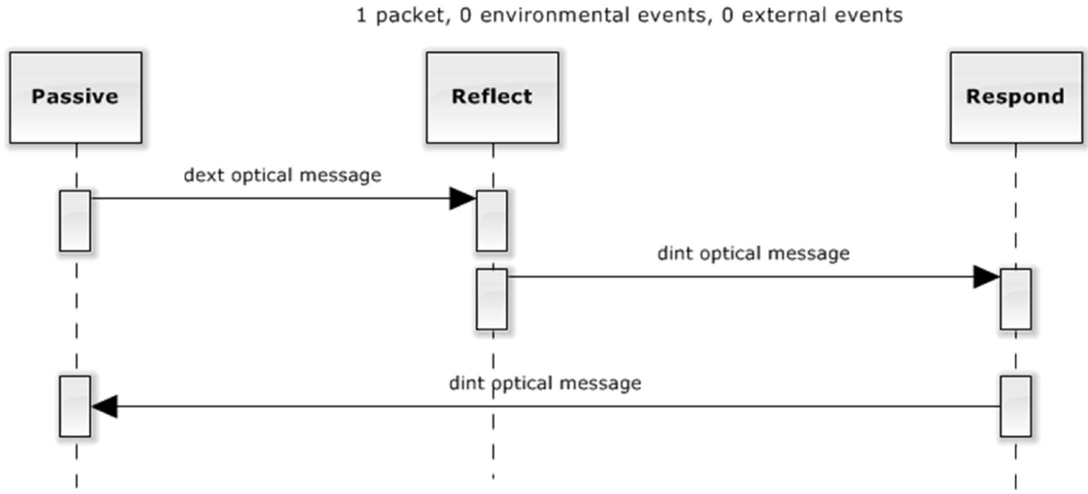


Figure 37. Case I sequence diagram.

E.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 15. *Case II state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	Interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

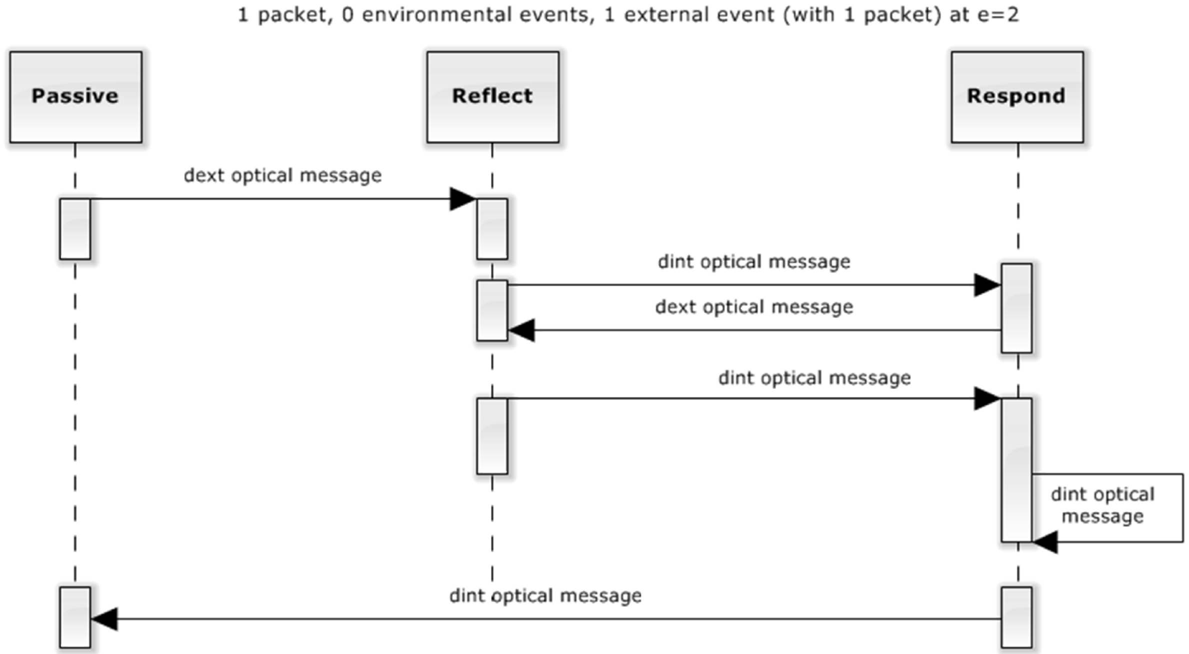


Figure 38. Case II sequence diagram.

E.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 16. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets

											(x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

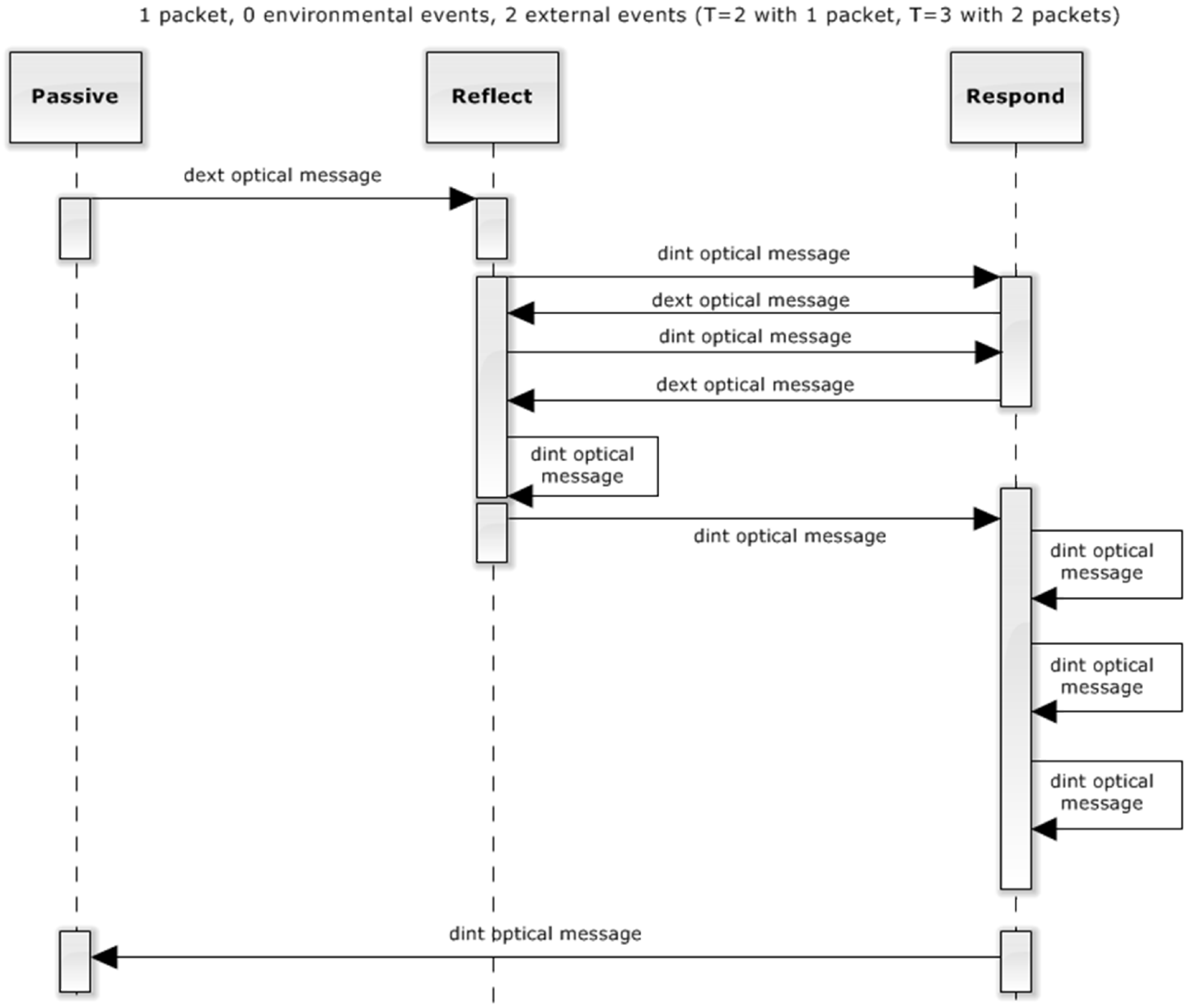


Figure 39. Case III sequence diagram.

E.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 17. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

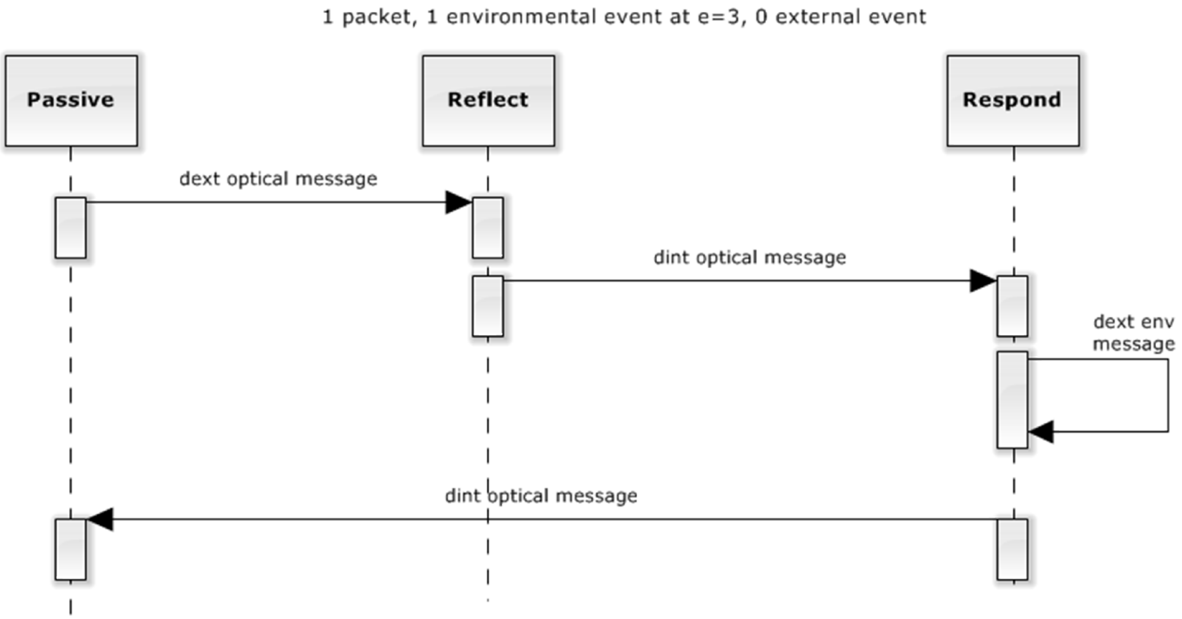


Figure 40. Case IV sequence diagram.

E.7 Beamsplitter Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.

- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, "store", temperature, "overtemp", "overpower", "interruptResond", queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level
"overpower" = flag variable set when device meets or exceeds damaged optical power level
"interruptRespond" = flag variable set when device is interrupted by an external event
Peak power = full width, half maximum power calculation of the pulse

For the beamsplitter we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+\rightarrow\infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{beamsplitter}$* **=** **(X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {"passive", "reflect", "respond"}

overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time delay$_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay$_i$) from *messagebag*

calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse

calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcHigh() = method that calculates the passed optical packet power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))

calcLow() = method that calculates the reflected optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))

calcForward() = method that calculates the forward direction optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReverse() = method that calculates the backward direction optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“OptIn_1”, “OptIn_2”, “OptIn_3”, “OptIn_4”, “EnvIn”} with
X_M = {(“OptIn_1”, V_{opt}), (“OptIn_2”, V_{opt}), (“OptIn_3”, V_{opt}), (“OptIn_4”, V_{opt}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“OptOut_1”, “OptOut_2”, “OptOut_3”, “OptOut_4”} with
Y_M = {(“OptOut_1”, $Y_{OptOut1}$), (“OptOut_2”, $Y_{OptOut2}$), (“OptOut_3”, $Y_{OptOut3}$), (“OptOut_4”, $Y_{OptOut4}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower* *interruptRespond,* *queue*} = {{“passive”, “reflect”, “respond”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x *V*}

External Transition Function:

```
δext(phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue, e, ((p_i,v_i),….
                                                                                       (p_n,v_n))) =
(“reflect”, 0, store, temperature, overtemp, overpower, queue.x1..xn)
  if phase = “passive” and p ∈ {“OptIn1”, “OptIn2”, “OptIn3”}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower = “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond , queue.x1..xn)
if phase = “respond” and p ∈ {“OptIn1”, “OptIn2”, “OptIn3”}
   update_delay(queue)
    for messagebag != null
```

```
        current = messagebag_first()
        if current.value.power > damaged.power
           overpower =  "Y"
        insert_event_q(current)
         remove_event_m(current)
     queue.current = queue_need_reflected()
     reflect = (queue.current.p), calcReflected(queue.current.v))
     mark_reflected(queue.current)
     interruptRespond= "Y"
    timeLeftRespond = timeLeftRespond - e

("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "passive" and p = "EnvIn"
     temperature = messagebag.temperature
     if temperature > damage.temp
        overtemp = "Y"

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                  queue.x1..xn)
    if phase = "respond" and p = "EnvIn"
      update_delay(queue)
      timeLeftRespond = time_delay- e
      temperature = messagebag.temperature
      if temperature > damage.temp
        overtemp = "Y"
      time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
("reflect", 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
   if phase = "reflect" and need.reflect != null
     need.reflect = queue_need_reflected()
     current = need.reflect
    reflect = (current.p), calcReflected(current.v))
    mark_reflected(current)

 ("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
   if phase = "reflect" and need.reflect = null
      need.reflect = queue_need_reflected()
      if interruptRespond = "N"
        current = queue_min()
        time.delay = current.time.delay
      if current.p  = “OptIn$_1$”              /* input port 1 – strong 4 weak 3 */
         new1 = (“OptOut$_3$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
         new2 = (“OptOut$_4$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
      else
         if current.p  = “OptIn$_2$”           /* input port 2 – strong 3 weak 4 */
            new1 = (“OptOut$_4$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
            new2 = (“OptOut$_3$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
      else
         if current.p  = “OptIn$_3$”           /* input port 3 – strong 2 weak 1 */
            new1 = (“OptOut$_1$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
            new2 = (“OptOut$_2$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
         else                                  /* input port 4 – strong 1 weak 2*/
            new1 = (“OptOut$_2$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
            new2 = (“OptOut$_1$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
     timeLeftRespond = propagation delay
     else
       time.delay = timeLeftRespond

 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if current.p  = “OptIn$_1$”              /* input port 1 – strong 4 weak 3 */
         new1 = (“OptOut$_3$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
         new2 = (“OptOut$_4$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
      else
         if current.p  = “OptIn$_2$”           /* input port 2 – strong 3 weak 4 */
            new1 = (“OptOut$_4$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
            new2 = (“OptOut$_3$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
      else
         if current.p  = “OptIn$_3$”           /* input port 3 – strong 2 weak 1 */
            new1 = (“OptOut$_1$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
            new2 = (“OptOut$_2$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
         else                                  /* input port 4 – strong 1 strong 2*/
            new1 = (“OptOut$_2$”,calcHigh(current.v, temperature, overtemp, peakpwr, overpwr))
            new2 = (“OptOut$_1$”,calcLow(current.v, temperature, overtemp, peakpwr, overpwr))
        interruptRespond= “N”

 (“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size  = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

$\lambda(phase, \sigma, store, temperature, overtemp, overpower) =$
  $(reflect.p, reflect.v)$
    if phase = “reflect”

  $(new1.p, new1.v)$
    if phase = “respond”

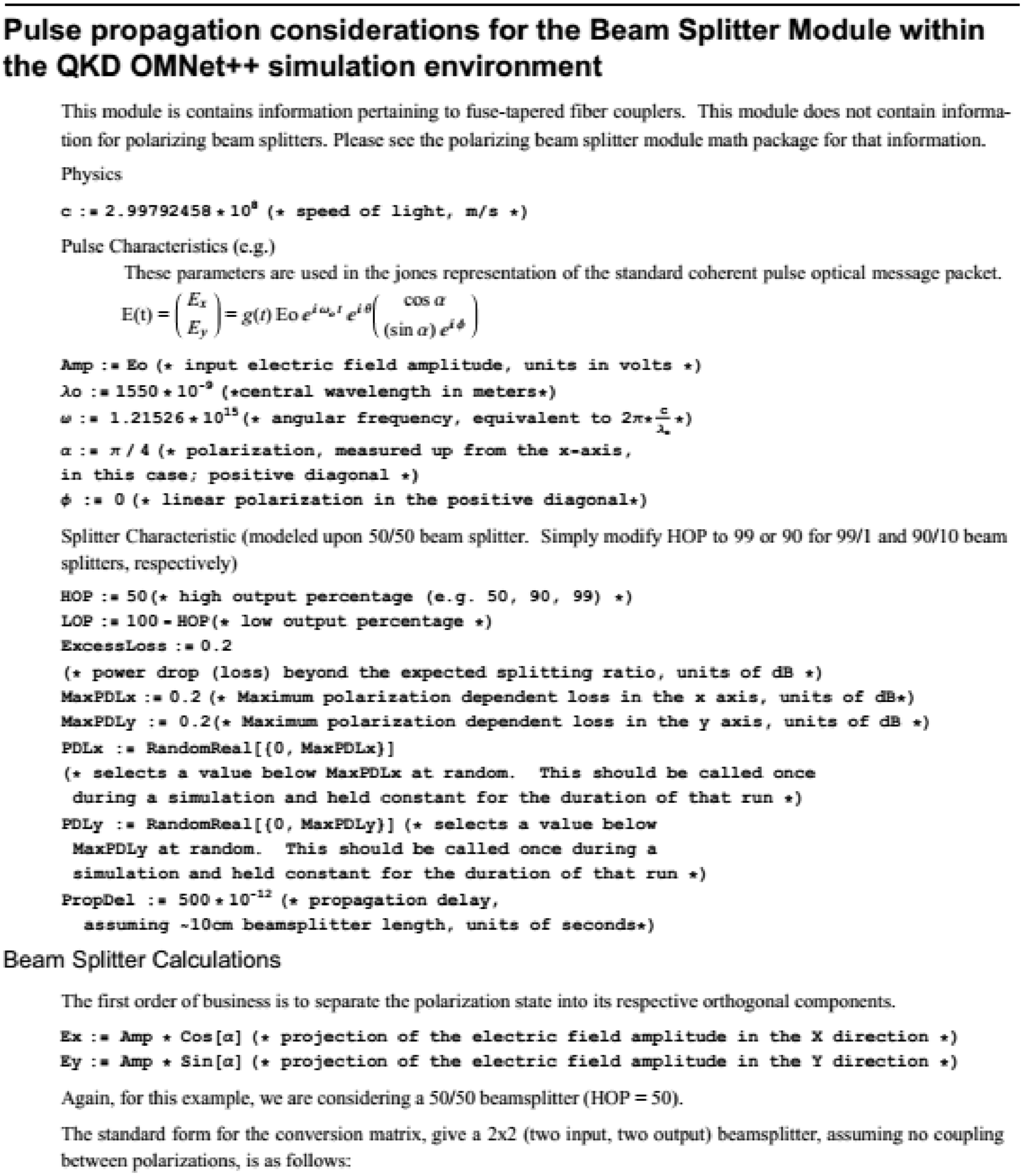
  $(new2.p, new2.v)$
    if phase = “respond”

  ∅ (null output)
    otherwise;

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower) = \sigma;$

E.8 Mathematical model



Pulse propagation considerations for the Beam Splitter Module within the QKD OMNet++ simulation environment

This module is contains information pertaining to fuse-tapered fiber couplers. This module does not contain information for polarizing beam splitters. Please see the polarizing beam splitter module math package for that information.

Physics

```
c := 2.99792458 * 10^8 (* speed of light, m/s *)
```

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\omega_o t}\,e^{i\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\phi} \end{pmatrix}$$

```
Amp := Eo (* input electric field amplitude, units in volts *)
λo := 1550 * 10^-9 (*central wavelength in meters*)
ω := 1.21526 * 10^15 (* angular frequency, equivalent to 2π*c/λo *)
α := π / 4 (* polarization, measured up from the x-axis,
in this case; positive diagonal *)
ϕ := 0 (* linear polarization in the positive diagonal*)
```

Splitter Characteristic (modeled upon 50/50 beam splitter. Simply modify HOP to 99 or 90 for 99/1 and 90/10 beam splitters, respectively)

```
HOP := 50(* high output percentage (e.g. 50, 90, 99) *)
LOP := 100 - HOP(* low output percentage *)
ExcessLoss := 0.2
(* power drop (loss) beyond the expected splitting ratio, units of dB *)
MaxPDLx := 0.2 (* Maximum polarization dependent loss in the x axis, units of dB*)
MaxPDLy := 0.2(* Maximum polarization dependent loss in the y axis, units of dB *)
PDLx := RandomReal[{0, MaxPDLx}]
(* selects a value below MaxPDLx at random.  This should be called once
 during a simulation and held constant for the duration of that run *)
PDLy := RandomReal[{0, MaxPDLy}] (* selects a value below
 MaxPDLy at random.  This should be called once during a
 simulation and held constant for the duration of that run *)
PropDel := 500 * 10^-12 (* propagation delay,
  assuming ~10cm beamsplitter length, units of seconds*)
```

Beam Splitter Calculations

The first order of business is to separate the polarization state into its respective orthogonal components.

```
Ex := Amp * Cos[α] (* projection of the electric field amplitude in the X direction *)
Ey := Amp * Sin[α] (* projection of the electric field amplitude in the Y direction *)
```

Again, for this example, we are considering a 50/50 beamsplitter (HOP = 50).

The standard form for the conversion matrix, give a 2x2 (two input, two output) beamsplitter, assuming no coupling between polarizations, is as follows:

$$\mathbf{M} := \frac{1}{10}\begin{pmatrix} \sqrt{\mathbf{LOP}} & i\sqrt{\mathbf{HOP}} & 0 & 0 \\ i\sqrt{\mathbf{HOP}} & \sqrt{\mathbf{LOP}} & 0 & 0 \\ 0 & 0 & \sqrt{\mathbf{LOP}} & i\sqrt{\mathbf{HOP}} \\ 0 & 0 & i\sqrt{\mathbf{HOP}} & \sqrt{\mathbf{LOP}} \end{pmatrix}$$

where the matrix M acts upon the input (ports 1 & 2) electric field values to yield the electric fields for the outputs (ports 3 & 4). If we assume that we have an input on port 1, leaving E2x and E2y to be zero, we have;

```
E1x := Ex
E1y := Ey
```

$$\begin{pmatrix} \mathbf{E3x} \\ \mathbf{E4x} \\ \mathbf{E3y} \\ \mathbf{E4y} \end{pmatrix} = \mathbf{M}.\begin{pmatrix} \mathbf{E1x} \\ \mathbf{E2x} \\ \mathbf{E1y} \\ \mathbf{E2y} \end{pmatrix} \ /.\ \mathbf{E2x} \to 0\ /.\ \mathbf{E2y} \to 0$$

$$\left\{\left\{\frac{\text{Amp Cos}[\alpha]}{\sqrt{2}}\right\}, \left\{\frac{i\ \text{Amp Cos}[\alpha]}{\sqrt{2}}\right\}, \left\{\frac{\text{Amp Sin}[\alpha]}{\sqrt{2}}\right\}, \left\{\frac{i\ \text{Amp Sin}[\alpha]}{\sqrt{2}}\right\}\right\}$$

We must now include loss. As loss is given in dB with relation to power, and we are dealing with the electric field here, some conversion is necessary.

$$\mathbf{E3xFin} = \mathbf{E3x} * \sqrt{10^{-\frac{\mathbf{ExcessLoss}}{10}}} * \sqrt{10^{-\frac{\mathbf{PDLx}}{10}}}$$

$$\mathbf{E4xFin} = \mathbf{E4x} * \sqrt{10^{-\frac{\mathbf{ExcessLoss}}{10}}} * \sqrt{10^{-\frac{\mathbf{PDLx}}{10}}}$$

```
0.478531 Eo

0.484021 i Eo
```

$$\mathbf{E3yFin} = \mathbf{E3y} * \sqrt{10^{-\frac{\mathbf{ExcessLoss}}{10}}} * \sqrt{10^{-\frac{\mathbf{PDLy}}{10}}}$$

$$\mathbf{E4yFin} = \mathbf{E4y} * \sqrt{10^{-\frac{\mathbf{ExcessLoss}}{10}}} * \sqrt{10^{-\frac{\mathbf{PDLy}}{10}}}$$

```
0.480859 Eo

0.484632 i Eo
```

To calculate the electric field amplitude of the output pulses we need to take the vector sum

$$\mathbf{E3Out} = \sqrt{\mathbf{E3xFin}^2 + \mathbf{E3yFin}^2}\ \ \mathbf{//\ FullSimplify}$$

$$0.678393\sqrt{\text{Eo}^2}$$

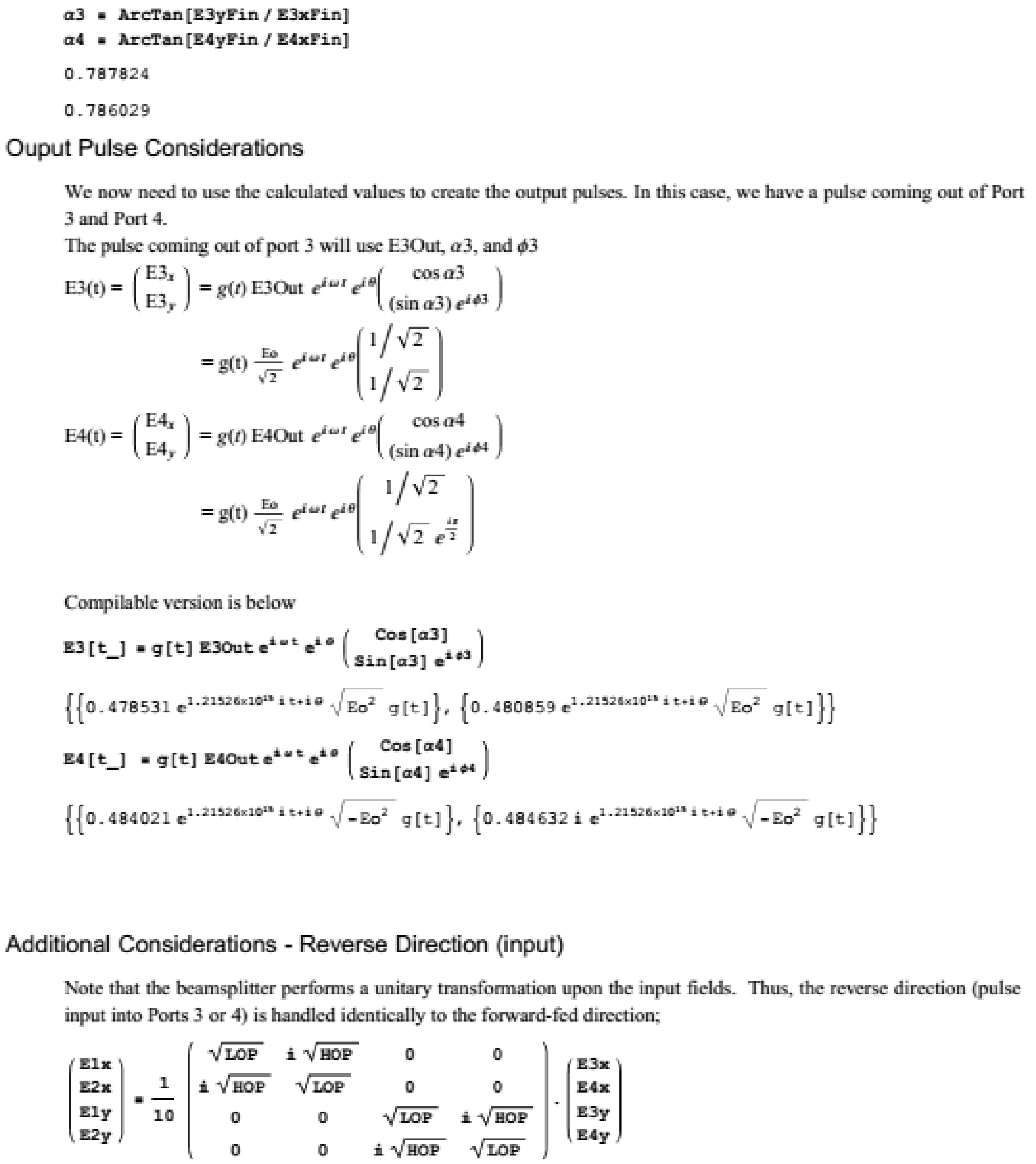
$$\mathbf{E4Out} = \sqrt{\mathbf{E4xFin}^2 + \mathbf{E4yFin}^2\ \mathbf{//\ FullSimplify}}$$

$$0.684941\sqrt{-\text{Eo}^2}$$

Note the negative above in the Eo. This is accounted for by altering the global phase (ϕ) of the output pulse at port 4 by $\pi/2$.

$$\boldsymbol{\phi}\mathbf{3} := 0$$
$$\boldsymbol{\phi}\mathbf{4} := \pi / 2$$

We also need to keep track of the output polarization(s).



```
α3 = ArcTan[E3yFin / E3xFin]
α4 = ArcTan[E4yFin / E4xFin]
```

```
0.787824

0.786029
```

Ouput Pulse Considerations

We now need to use the calculated values to create the output pulses. In this case, we have a pulse coming out of Port 3 and Port 4.
The pulse coming out of port 3 will use E3Out, $\alpha 3$, and $\phi 3$

$$\text{E3(t)} = \begin{pmatrix} \text{E3}_x \\ \text{E3}_y \end{pmatrix} = g(t)\ \text{E3Out}\ e^{i\omega t}\, e^{i\theta} \begin{pmatrix} \cos\alpha 3 \\ (\sin\alpha 3)\, e^{i\phi 3} \end{pmatrix}$$

$$= \text{g(t)}\ \frac{\text{Eo}}{\sqrt{2}}\ e^{i\omega t}\, e^{i\theta} \begin{pmatrix} 1\big/\sqrt{2} \\ 1\big/\sqrt{2} \end{pmatrix}$$

$$\text{E4(t)} = \begin{pmatrix} \text{E4}_x \\ \text{E4}_y \end{pmatrix} = g(t)\ \text{E4Out}\ e^{i\omega t}\, e^{i\theta} \begin{pmatrix} \cos\alpha 4 \\ (\sin\alpha 4)\, e^{i\phi 4} \end{pmatrix}$$

$$= \text{g(t)}\ \frac{\text{Eo}}{\sqrt{2}}\ e^{i\omega t}\, e^{i\theta} \begin{pmatrix} 1\big/\sqrt{2} \\ 1\big/\sqrt{2}\ e^{\frac{i\pi}{2}} \end{pmatrix}$$

Compilable version is below

$$\text{E3[t\_] = g[t] E3Out}\ e^{i\omega t}\, e^{i\theta} \begin{pmatrix} \text{Cos}[\alpha 3] \\ \text{Sin}[\alpha 3]\ e^{i\phi 3} \end{pmatrix}$$

$$\left\{\left\{0.478531\, e^{1.21526\times 10^{15}\, i\, t + i\,\theta}\, \sqrt{\text{Eo}^2}\ \text{g[t]}\right\}, \left\{0.480859\, e^{1.21526\times 10^{15}\, i\, t + i\,\theta}\, \sqrt{\text{Eo}^2}\ \text{g[t]}\right\}\right\}$$

$$\text{E4[t\_] = g[t] E4Out}\ e^{i\omega t}\, e^{i\theta} \begin{pmatrix} \text{Cos}[\alpha 4] \\ \text{Sin}[\alpha 4]\ e^{i\phi 4} \end{pmatrix}$$

$$\left\{\left\{0.484021\, e^{1.21526\times 10^{15}\, i\, t + i\,\theta}\, \sqrt{-\text{Eo}^2}\ \text{g[t]}\right\}, \left\{0.484632\, i\, e^{1.21526\times 10^{15}\, i\, t + i\,\theta}\, \sqrt{-\text{Eo}^2}\ \text{g[t]}\right\}\right\}$$

Additional Considerations - Reverse Direction (input)

Note that the beamsplitter performs a unitary transformation upon the input fields. Thus, the reverse direction (pulse input into Ports 3 or 4) is handled identically to the forward-fed direction;

$$\begin{pmatrix} \text{E1x} \\ \text{E2x} \\ \text{E1y} \\ \text{E2y} \end{pmatrix} = \frac{1}{10} \begin{pmatrix} \sqrt{\text{LOP}} & i\sqrt{\text{HOP}} & 0 & 0 \\ i\sqrt{\text{HOP}} & \sqrt{\text{LOP}} & 0 & 0 \\ 0 & 0 & \sqrt{\text{LOP}} & i\sqrt{\text{HOP}} \\ 0 & 0 & i\sqrt{\text{HOP}} & \sqrt{\text{LOP}} \end{pmatrix} . \begin{pmatrix} \text{E3x} \\ \text{E4x} \\ \text{E3y} \\ \text{E4y} \end{pmatrix}$$

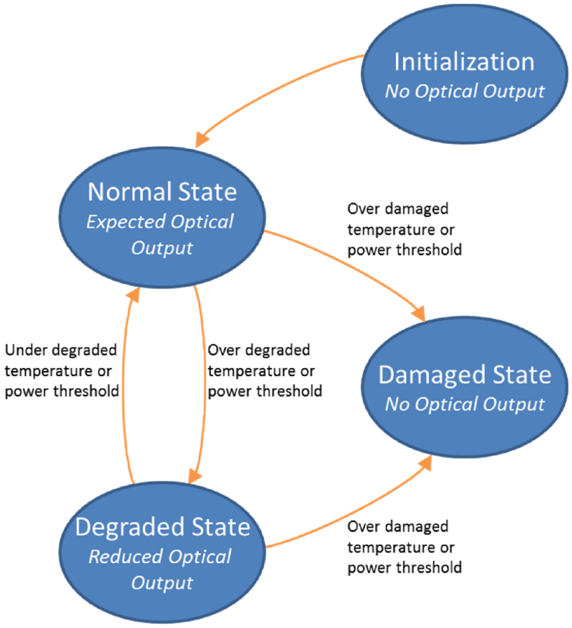
E.9 Component Use Cases

E.9.1 Respond to an Optical Packet in the Beamsplitter

Optical packet arrives at beamsplitter. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the beamsplitter. Records overpower condition, if applicable. Remove the optical packet from the queue and split the packet into transmitted and reflected packets. Calculate the attenuated optical output signals based on the input signal, the input optical port and the current component state. Propagate the attenuated optical output signal out of the component optical port based on the input port.

- Identified Alternative Uses Cases
    - React to an environmental message

- Assumptions
    - Component has completed initialization sequence at least once
    - Reflections are not affected by component state
    - Incoming electrical signals are not affected by component state



Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 41. Component states.

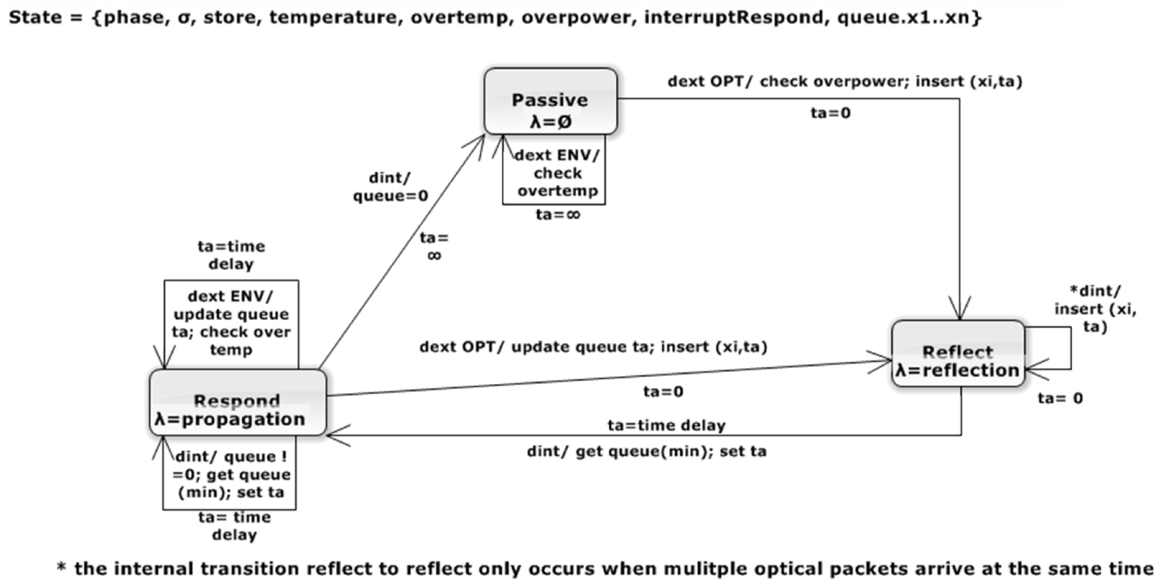

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/
check
overtemp
ta=∞
dint/
queue=0
ta=
∞
ta=time
delay
dext ENV/
update queue
ta; check over
temp
*dint/
insert (xi,
ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
ta=0
ta= 0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue !
=0; get queue
(min); set ta
ta= time
delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 42. Beamsplitter phase transition diagram.

E.9.2 End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.
- Optical packet attenuated properly to the limit of accuracy.
- Optical packet propagated out the correct port at the correct time.

E.9.3 Respond to an Environmental Packet in the Beamsplitter

Environmental packet arrives at the beamsplitter. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
  - None

E.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly

- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

E.10 Beamsplitter Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical

SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Beamsplitter Test Cases.*

		Inject Port						Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Env**	Notes	opt #	**env #**
Passive	1	1	0	0	0	0	single	1	0
	2	0	1	0	0	0	single	2	0
	3	0	0	1	0	0	single	3	0
	4	0	0	0	1	0	single	4	0
	5	0	0	0	0	1	single	4	1
	6	1	1	1	1	0	same time	8	1
	7	1	1	1	1	0	differ time	12	1
	8	1	1	1	1	1	same time	16	2
	9	1	1	1	1	1	differ time	20	3
	10	0	1	0	0	1	same time	21	4
	11	0	1	0	0	1	differ time	22	5
	12	1	0	0	0	1	same time	23	6
	13	1	0	0	0	1	differ time	24	7
	14	0	0	1	0	1	same time	25	8
	15	0	0	1	0	1	differ time	26	9
	16	0	0	0	1	1	same time	27	10
	17	0	0	0	1	1	differ time	28	11
	20	2	0	0	0	0	same time	30	11
	21	0	2	0	0	0	same time	32	11
	22	0	0	2	0	0	same time	34	11
	23	0	0	0	2	0	same time	36	11
	24	2	2	2	2	0	same time	44	11
	25	2	2	2	2	0	differ time	52	11
	26	2	2	2	2	1	same time	60	12
	27	2	2	2	2	1	differ time	68	13
	28	0	2	0	0	1	same time	70	14
	29	0	2	0	0	1	differ time	72	15
	30	2	0	0	0	1	same time	74	16
	31	2	0	0	0	1	differ time	76	17
	32	0	0	2	0	1	same time	78	18
	33	0	0	2	0	1	differ time	80	19
	34	0	0	0	2	1	same time	82	20
	35	0	0	0	2	1	differ time	84	21
totals		21	21	21	21	21	84		
Respond	41	2	0	0	0	0	single	86	21

42	0	2	0	0	0	single	88	21
43	0	0	2	0	0	single	90	21
44	0	0	0	2	0	single	92	21
45	1	0	0	0	1	single	93	22
46	2	1	1	1	0	same time	98	22
47	2	1	1	1	0	differ time	103	22
48	2	1	1	1	1	same time	108	23
49	2	1	1	1	1	differ time	113	24
50	0	2	0	0	1	same time	115	25
51	0	2	0	0	1	differ time	117	26
52	2	0	0	0	1	same time	119	27
53	2	0	0	0	1	differ time	121	28
54	0	0	2	0	1	same time	123	29
55	0	0	2	0	1	differ time	125	30
56	0	0	0	2	1	same time	127	31
57	0	0	0	2	1	differ time	129	32
60	3	0	0	0	0	same time	132	32
61	0	3	0	0	0	same time	135	32
62	0	0	3	0	0	same time	138	32
63	0	0	0	3	0	same time	141	32
64	3	2	2	2	0	same time	150	32
65	3	2	2	2	0	differ time	159	32
66	3	2	2	2	1	same time	168	33
67	3	2	2	2	1	differ time	177	34
68	0	3	0	0	1	same time	180	35
69	0	3	0	0	1	differ time	183	36
70	3	0	0	0	1	same time	186	37
71	3	0	0	0	1	differ time	189	38
72	0	0	3	0	1	same time	192	39
73	0	0	3	0	1	differ time	195	40
74	0	0	0	3	1	same time	198	41
75	0	0	0	3	1	differ time	201	42
totals	36	27	27	27	21	117		
TC1	1	0	0	0	2	single	202	44
TC2	1	0	0	0	2	single	203	46
TC3	1	0	0	0	2	single	204	48
TC4	1	0	0	0	2	single	205	50
TC5	1	0	0	0	2	single	206	52
TC6	1	0	0	0	2	single	207	54
totals	6	0	0	0	12	6		

E.11 References

OZOptics. (2013). Beam splitters/combiners. Retrieved September 25, 2013, from http://www.ozoptics.com/ALLNEW_PDF/DTS0095.pdf

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

Appendix F - Circulator

F.1 Device Description:

The circulator is an optical device allows light to pass through in one direction. Light entering port one exits port two with minimal attenuation but is highly attenuated leaving port three. Light entering port two exits port three with minimal attenuation and is highly attenuated leaving port one. A “full” circulator allows light to enter port three and pass on to port one with minimal attenuation but heavily attenuating port two. A “quasi” circulator highly attenuates any light entering port three at ports one and two. Circulators are used in multiplexers, bi-directional pumps and chromatic dispersion compensation devices (ThorLabs, 2013). See Figure 1 for an example of a quasi-circulator.

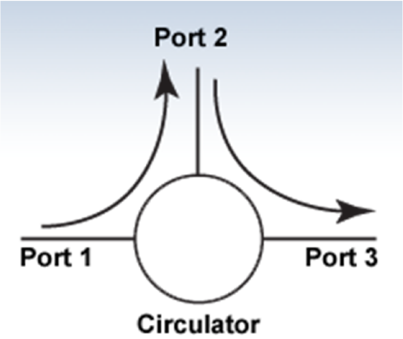


Figure 43. View of a three port circulator (ThorLabs, 2013).

A quasi-circulator can be made from a polarizing beam splitter, a Faraday rotator and a half-wave plate in line. Any light travelling in the forward direction through the circulator has its polarization changed by 90° as it passes through the Faraday rotator and half-wave plate. This allows for light to pass from port one to port two and from port two to port three. Light from port three to port one is directed into the circulator housing by the polarizing beam splitter. Any light travelling backwards through the ports is rotated then directed to the housing by the polarizing beam splitter, but some light does leak out the incorrect ports. For a four-port device, the Faraday rotator and half-wave plate are sandwiched between two polarizing beam splitters.

Circulators are non-reciprocal optical devices because the changes to the properties of light are not reversed by travelling backwards through the device (Saleh & Teich, 1991). Although there are polarization-independent and polarization-dependent types of circulators, this research will consider the polarization-independent version only, as the polarization dependent version is used in highly limited applications (per conversation with SME). The model developed here will be for the full-circulator, although quasi-circulators are more commonly used, as the full-circulator is the more general model.

The Circulator is a bidirectional optical component with three optical ports. Optical signals arriving at the input port are propagated to next port in sequence after a defined propagation delay and the polarizing material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the Circulator may become permanently damaged which changes its propagation characteristics. Similarly, the Circulator is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the Circulator may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the Circulator is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the Circulator. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the Circulator to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the Circulator using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the Circulator. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

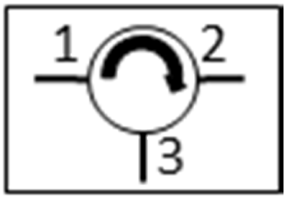


Figure 44. Symbol for the 3-port typical circulator in the QKD system architecture.

F.2 Circulator Conceptual Model

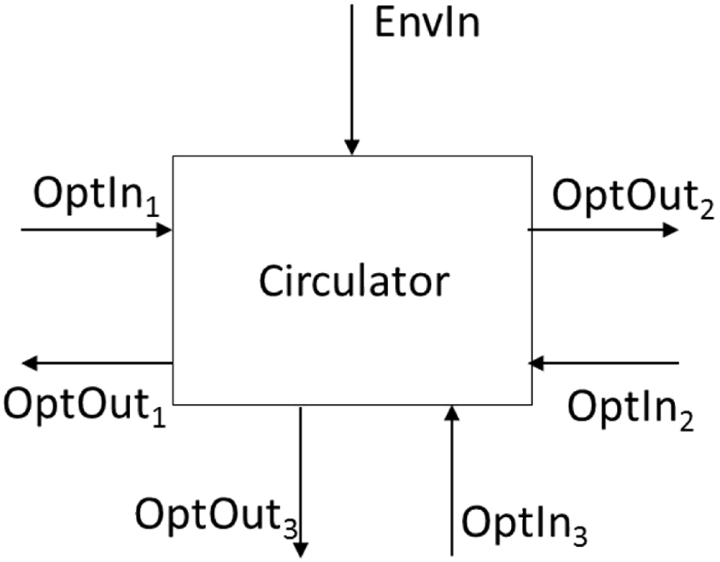


Figure 45. Circulator conceptual model.

The conceptual model for a Circulator consists of three optical input ports {OptIn$_1$, OptIn$_2$, OptIn$_3$}, three optical output ports {OptOut$_1$, OptOut$_2$, OptOut$_2$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the Circulator. In comparison to the Circulator symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the Circulator, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn$_1$ in Fig. 3 will instantaneously generate a reflected emitting out of OptOut$_1$.

The Circulator calculates changes to the power (attenuation) and polarization of any packet coming through an optical port after a time equaling the propagation delay of the module.

The packet is calculated at full power minus some small amount to account for attenuation through the device if passing through to the correct port or heavily attenuated if passing to an incorrect port (for example, passing from port three to port one in a quasi-circulator). The model handles this undesired throughput by splitting each incoming optical packet into a 'strong' packet (one that is output through the correct port) and a 'weak' packet (one that is output through the incorrect port) and injecting these packets into the queue. Each of these entries are a (port, value) pair, just as any other entry into the queue. Additionally, every packet is rotated by 90° due to the effects of the Faraday rotator and half-wave plate as it passes through the device and this effect is applied by the 'strong' and 'weak' functions.

The Circulator must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the Circulator can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

F.3 Mathematical Model

For a detailed mathematical description of the Circulator, refer to Section 4.8 which contains the Mathematica worksheet provided by the optical physics SME.

F.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the Circulator.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Split the incoming optical packet into two entries in the queue.
- Update the values for one queue entry as a 'strong' optical signal based on the characteristics of the circulator, the original values of the input optical signal and the current environment and set the correct output port.
- Update the values for the other queue entry as a 'weak' optical signal based on the characteristics of the circulator, the original values of the input optical signal and the current environment and set the correct output port.
- After the propagation time has elapsed, send the output signal out of the optical output port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

F.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the Circulator in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the Circulator at the same time.

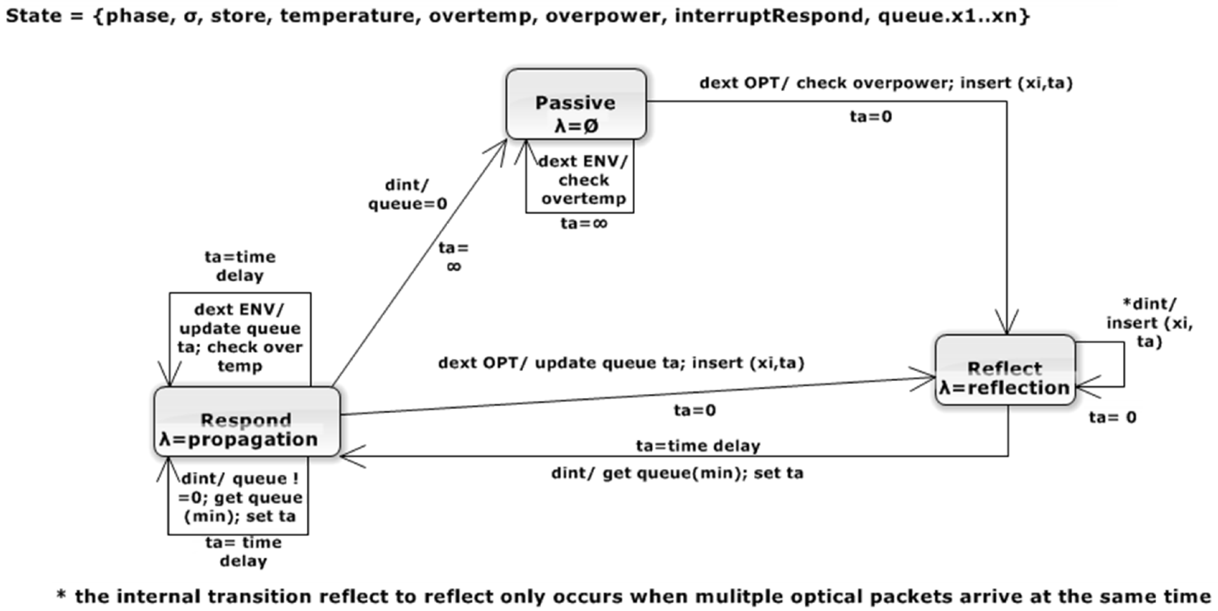

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
ta=0
ta= 0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 46. Circulator phase transition diagram.

F.6 Event-Trace Diagram

This section shows various examples of packets entering the Circulator. The tables list the states the circulator proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the Circulator, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

F.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 18. *Case I state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	

5	s3	entry	passive	inf	x1	c	n	n	n	null	

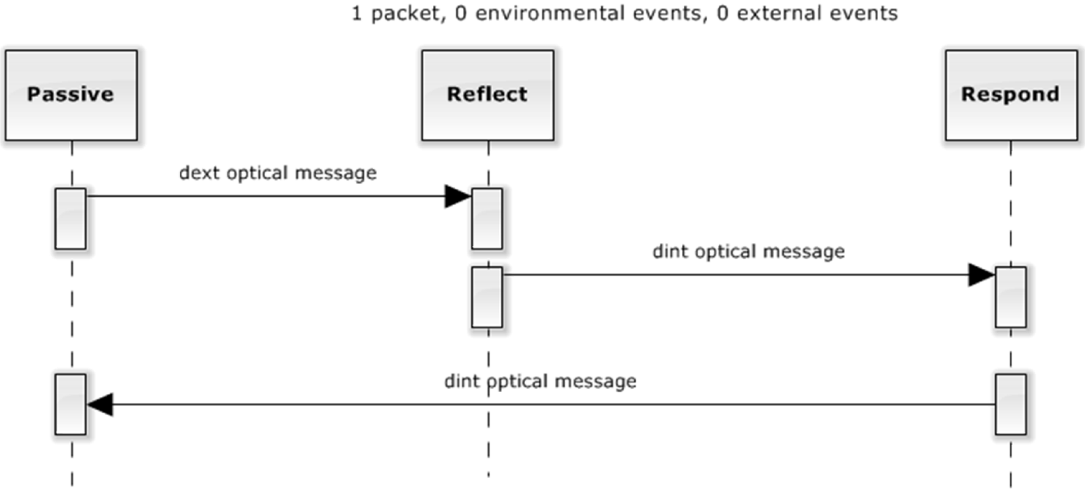


Figure 47. Case I sequence diagram.

F.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 19. *Case II state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**Interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

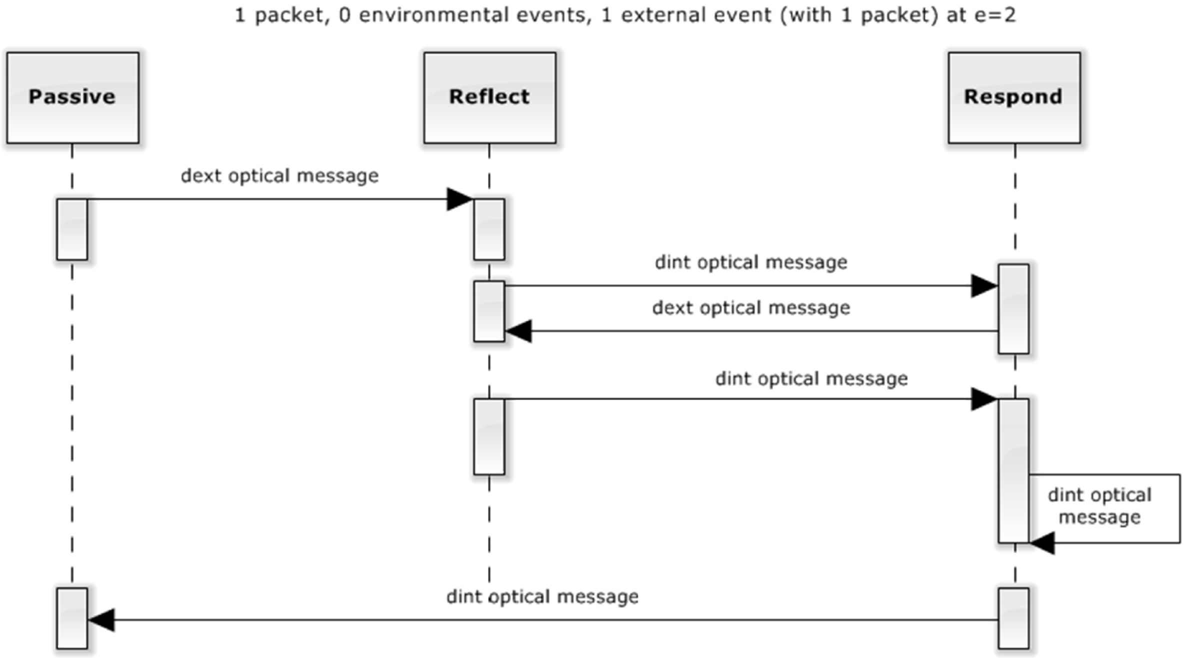


Figure 48. Case II sequence diagram.

F.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 20. *Case III state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	

3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

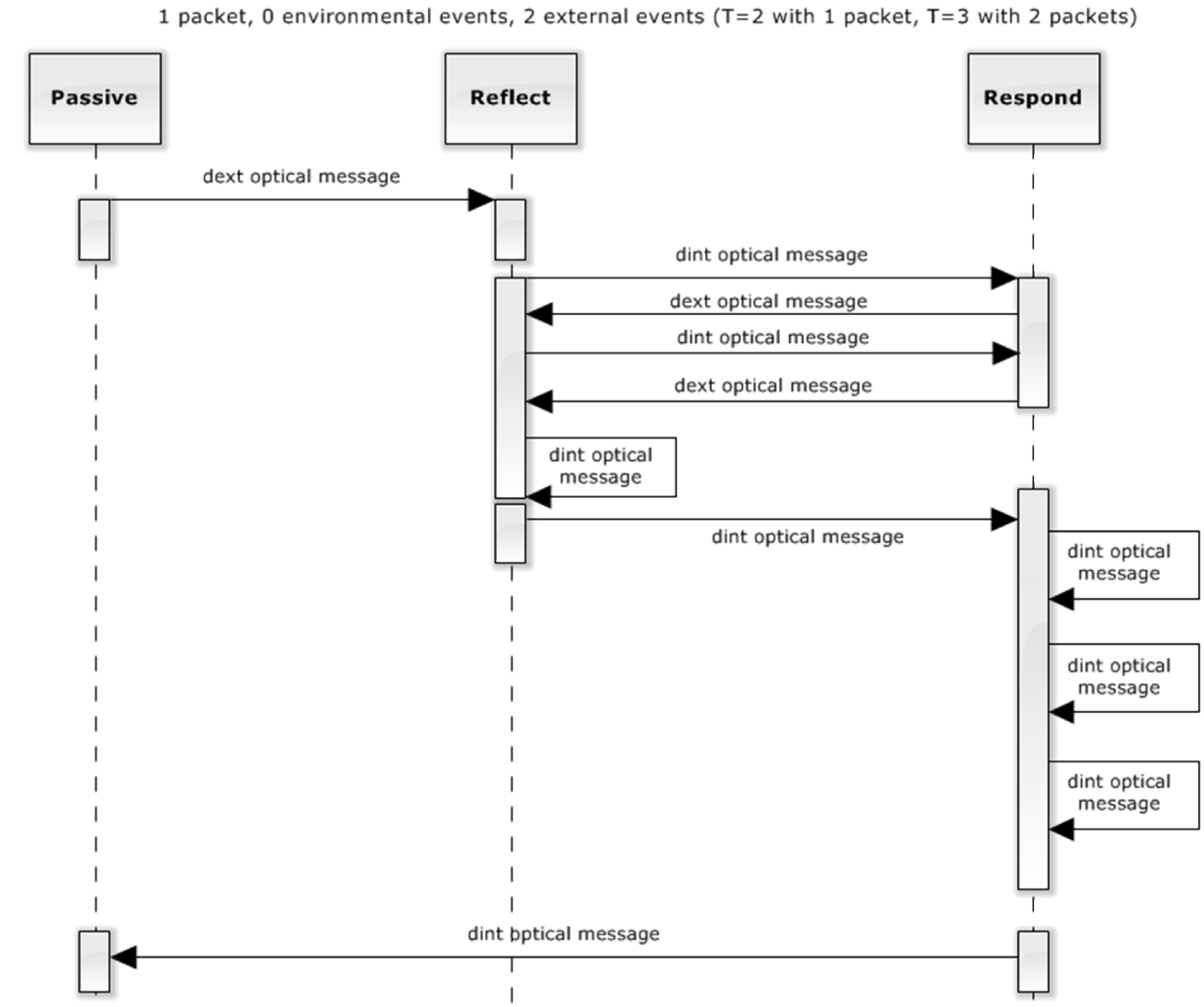


Figure 49. Case III sequence diagram.

F.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 21. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

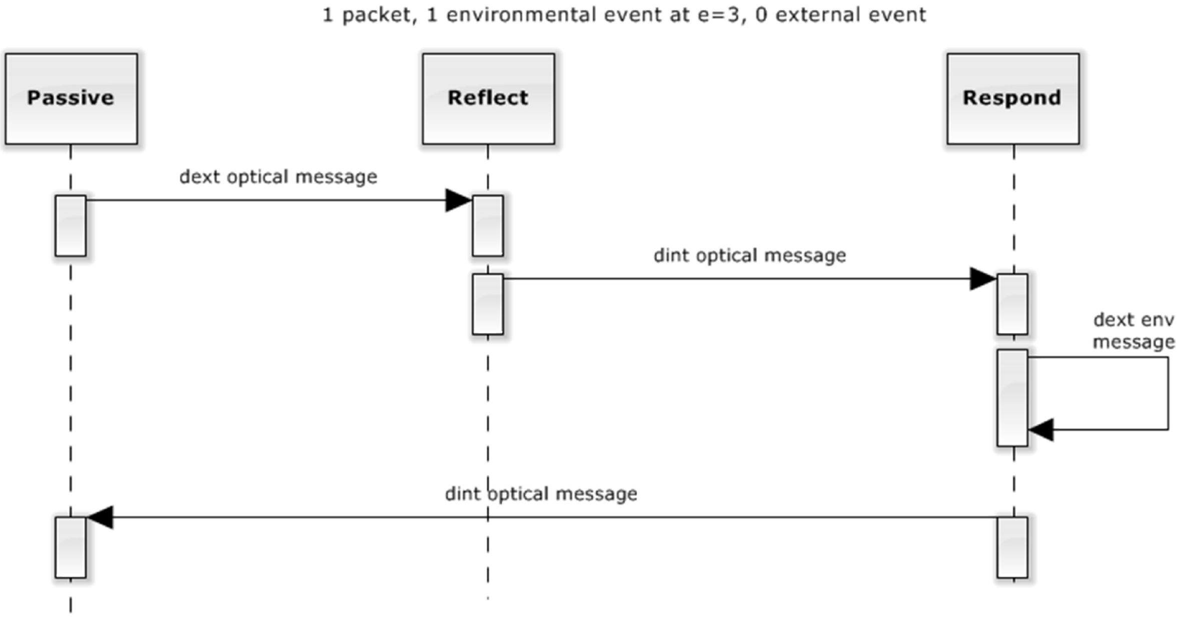


Figure 50. Case IV sequence diagram.

F.7 Circulator Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.

- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, "store", temperature, "overtemp", "overpower", "interruptRespond", queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level
"overpower" = flag variable set when device meets or exceeds damaged optical power level
"interruptRespond" = flag variable set when device is interrupted by an external event
Peak power = full width, half maximum power calculation of the pulse

For the Circulator we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

DEVS$_{\boldsymbol{Circulator}}$ **=** $\boldsymbol{(X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {"passive", "reflect", "respond"}

overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
new1= variable to hold 1st output values
new2 = variable to hold 2nd output values
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time $delay_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its x (p,v) values into the variable *store*

InPorts = {"OptIn$_1$", "OptIn$_2$", "OptIn$_3$", "EnvIn"} with
X_M = {("OptIn$_1$", V_{opt}), ("OptIn$_2$", V_{opt}), ("OptIn$_3$", V_{opt}), ("EnvIn", V_{env})} is the set of input ports and values.

OutPorts = {"OptOut$_1$", "OptOut$_2$", "OptOut$_3$"} with
Y_M = {("OptOut$_1$", $Y_{OptOut1}$), ("OptOut$_2$", $Y_{OptOut2}$), ("OptOut$_3$", $Y_{OptOut3}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, queue*} = {{"passive", "reflect", "respond"} x R_0^+ x V x R x {"Y", "N"} x {"Y","N"} x {"Y","N"} x V}

External Transition Function:

δ_{ext}(*phase*, σ, *store, temperature*, *overtemp*, *overpower, interruptRespond, queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =
("reflect", 0, *store, temperature*, *overtemp*, *overpower, interruptRespond, queue.x1..xn*)

```
if phase = "passive" and p ∈ {"OptIn_1", "OptIn_2", "OptIn_3"}
    for messagebag != null
        current = messagebag_first()
         if current.value.power > damaged.power
            overpower = "Y"
         insert_event_q(current)
         remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = "N"
```

```
(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond ,queue.x1..xn)
if phase = “respond” and p ∈ {“$OptIn_1$”, “$OptIn_2$”, “OptIn3”}
    update_delay(queue)
      for messagebag != null
        current = messagebag_first()
        if current.value.power > damaged.power
          overpower =  “Y”
        insert_event_q(current)
         remove_event_m(current)
     queue.current = queue_need_reflected()
     reflect = (queue.current.p), calcReflected(queue.current.v))
     mark_reflected(queue.current)
     interruptRespond= “Y”
    timeLeftRespond = timeLeftRespond - e

(“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “passive” and p = “EnvIn”
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                   queue.x1..xn)
    if phase = “respond” and p = “EnvIn”
      update_delay(queue)
      timeLeftRespond = time_delay- e
      temperature = messagebag.temperature
      if temperature > damage.temp
        overtemp = “Y”
      time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
(“reflect”, 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
   if phase = “reflect” and need.reflect != null
     need.reflect = queue_need_reflected()
     current = need.reflect
    reflect = (current.p), calcReflected(current.v))
    mark_reflected(current)

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
```

```
  if phase = “reflect” and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = “N”
       current = queue_min()
       time.delay = current.time.delay
   if current.p  = “OptIn$_1$”               /* input port 1 – strong 2 weak 3 */
    new1 = (“OptOut2”,calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
    new2 = (“OptOut3”,calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
   else
     if current.p  = “OptIn$_2$”             /* input port 2 – strong 3 weak 1 */
       new1 = (“OptOut3”,calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
       new2 = (“OptOut1”,calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     else                                    /* input port 3 – strong 1 weak 2*/
       new1 = (“OptOut1”,calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
       new2 = (“OptOut2”,calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
   timeLeftRespond = propagation delay
  else
     time.delay = timeLeftRespond

 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = “respond” and size > 0
     update_delay(queue)
     size= queue_size()
     current = queue_min()
     time.delay = current.time.delay
     if current.p  = “OptIn$_1$”             /* input port 1 – strong 2 weak 3 */
      new1 = (“OptOut2”,calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
      new2 = (“OptOut3”,calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     else
       if current.p  = “OptIn$_2$”           /* input port 2 – strong 3 weak 1 */
         new1 = (“OptOut3”,calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
         new2 = (“OptOut1”,calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
       else                                  /* input port 3 – strong 1 weak 2*/
         new1 = (“OptOut1”,calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
         new2 = (“OptOut2”,calcWeak (current.v, temperature, overtemp, peakpwr, overpwr))
      interruptRespond= “N”

 (“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = “respond” and size  = 0
     size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

λ(*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, queue*) =

(*reflect.p, reflect.v*)
if phase = “reflect”

(*new1.p*, *new1.v*)
if phase = “respond”

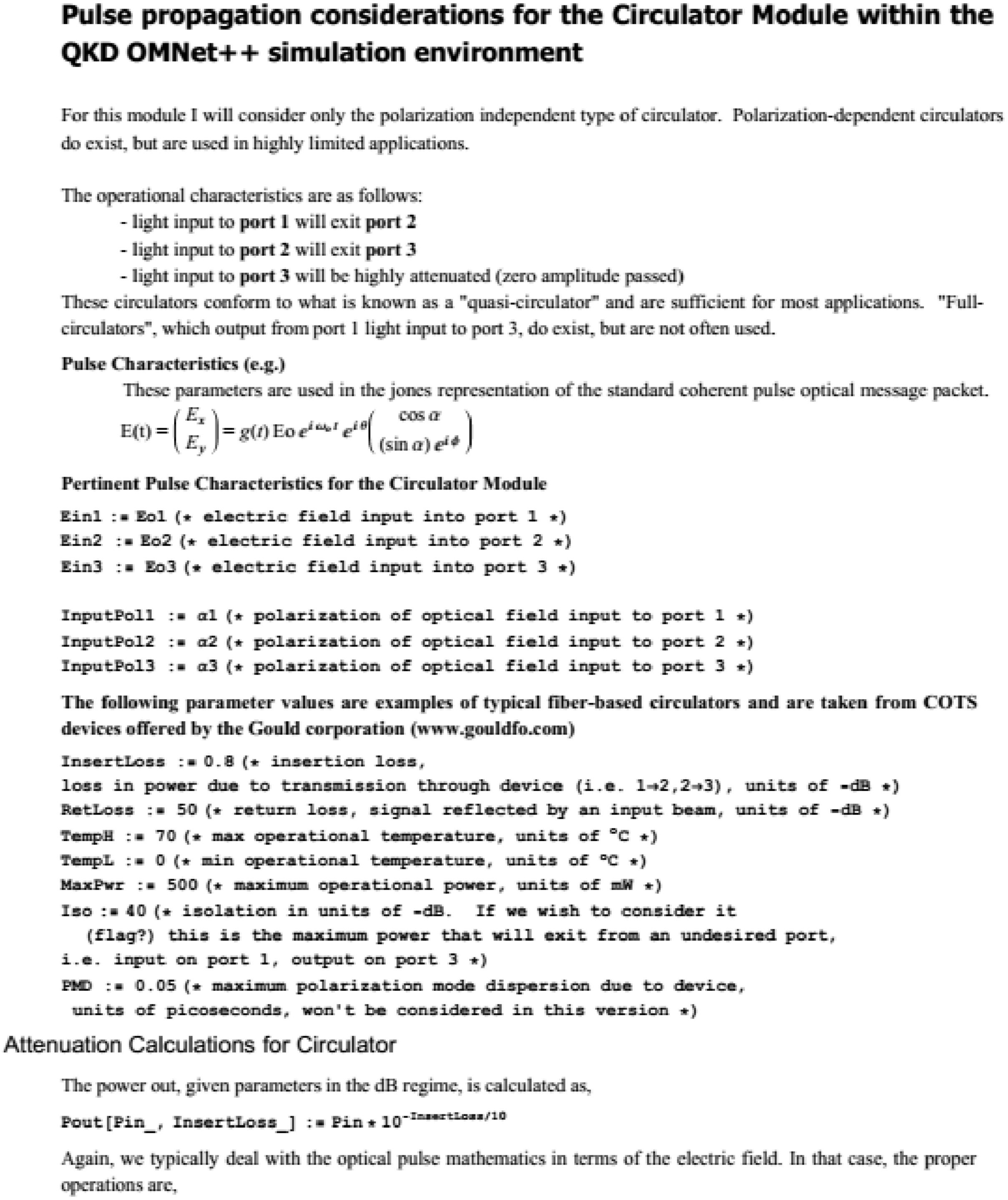
(*new2.p*, *new2.v*)
if phase = “respond”

∅ (null output)
otherwise;

Time advance Function:

ta(*phase*, σ, *store*, *temperature*, *overtemp*, *overpower*) = σ;

F.8 Mathematical model



Pulse propagation considerations for the Circulator Module within the QKD OMNet++ simulation environment

For this module I will consider only the polarization independent type of circulator. Polarization-dependent circulators do exist, but are used in highly limited applications.

The operational characteristics are as follows:
- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 3**
- light input to **port 3** will be highly attenuated (zero amplitude passed)

These circulators conform to what is known as a "quasi-circulator" and are sufficient for most applications. "Full-circulators", which output from port 1 light input to port 3, do exist, but are not often used.

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\,\omega_o t}\,e^{i\,\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Circulator Module

```
Ein1 := Eo1 (* electric field input into port 1 *)
Ein2  := Eo2 (* electric field input into port 2 *)
Ein3  := Eo3 (* electric field input into port 3 *)

InputPol1 := α1 (* polarization of optical field input to port 1 *)
InputPol2 := α2 (* polarization of optical field input to port 2 *)
InputPol3 := α3 (* polarization of optical field input to port 3 *)
```

The following parameter values are examples of typical fiber-based circulators and are taken from COTS devices offered by the Gould corporation (www.gouldfo.com)

```
InsertLoss := 0.8 (* insertion loss,
loss in power due to transmission through device (i.e. 1→2,2→3), units of -dB *)
RetLoss := 50 (* return loss, signal reflected by an input beam, units of -dB *)
TempH := 70 (* max operational temperature, units of °C *)
TempL := 0 (* min operational temperature, units of °C *)
MaxPwr := 500 (* maximum operational power, units of mW *)
Iso := 40 (* isolation in units of -dB.  If we wish to consider it
  (flag?) this is the maximum power that will exit from an undesired port,
i.e. input on port 1, output on port 3 *)
PMD := 0.05 (* maximum polarization mode dispersion due to device,
 units of picoseconds, won't be considered in this version *)
```

Attenuation Calculations for Circulator

The power out, given parameters in the dB regime, is calculated as,

```
Pout[Pin_, InsertLoss_] := Pin * 10^(-InsertLoss/10)
```

Again, we typically deal with the optical pulse mathematics in terms of the electric field. In that case, the proper operations are, Eout2[Ein1_, InsertLoss_] := Ein1 * $\sqrt{10^{-\text{InsertLoss}/10}}$ (* case: optical input on port 1 *)

Eout3[Ein2_, InsertLoss_] := Ein2 * $\sqrt{10^{-\text{InsertLoss}/10}}$ (* case: optical input on port 2 *)

Eout2[Ein3_, InsertLoss_] := Ein3 * 0.0 (* case: optical input on port 3 *)

If we wish to flag the circulator to include **undesired port throughput**, the following operations would hold true,

Eout3[Ein1_, Iso_] := Ein1 * $\sqrt{10^{-\text{Iso}/10}}$
(* case: optical input on port 1, undesired output port 3 *)

Eout1[Ein2_, Iso_] := Ein2 * $\sqrt{10^{-\text{Iso}/10}}$
(* case: optical input on port 2, undesired output port 1 *)

Eout2[Ein3_, Iso_] := Ein3 * $\sqrt{10^{-\text{Iso}/10}}$
(* case: optical input on port 3, undesired output port 2 *)

Eout1[Ein3_, Iso_] :=
Ein3 * $\sqrt{10^{-\text{Iso}/10}}$ (* case: optical input on port 3, undesired output port 1 *)

If we wish to flag the circulator to include **undesired return (reflected)** messages, the following operations would hold true,

Eout1[Ein1_, RetLoss_] := Ein1 * $\sqrt{10^{-\text{RetLoss}/10}}$

Eout2[Ein2_, RetLoss_] := Ein2 * $\sqrt{10^{-\text{RetLoss}/10}}$

Eout3[Ein3_, RetLoss_] := Ein3 * $\sqrt{10^{-\text{RetLoss}/10}}$

Polarizaion Calculations

Due to the nature/function of a polarization independent fiber-based circulator, the beam polarization is rotated by 90° in from port 1 to port 2, and again from port 2 to port 3. This rotation is in the same "direction", regardless of which path (1->2, 2->3) the light takes.

OutputPol2[InputPol1_] := InputPol1 - $\frac{\pi}{2}$

OutputPol3[InputPol2_] := InputPol2 - $\frac{\pi}{2}$

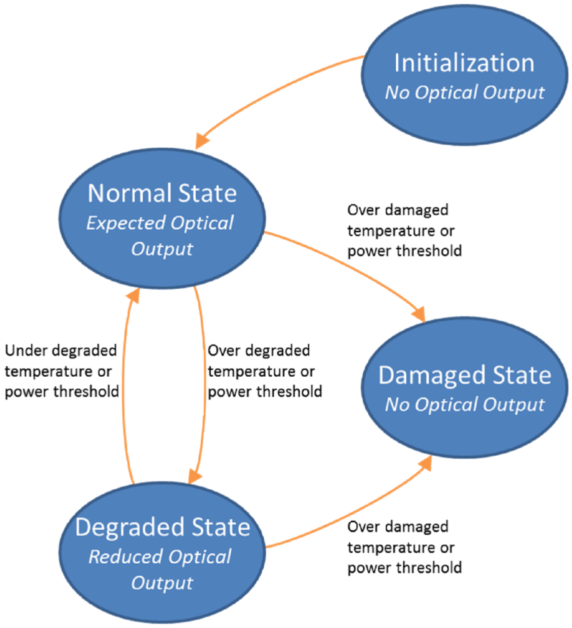
F.9 Component Use Cases

F.9.1 Respond to an Optical Packet in the Circulator

Optical packet arrives at the circulator. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the circulator. Records overpower condition, if applicable. Remove the optical packet from the queue and split the optical packet into strong and weak pulse packets. Calculate the attenuated optical output signal based on the input signal, the strength of the packet and the current component state. Propagate the attenuated optical output signal out of the component optical port based on the input port and component design.

- Identified Alternative Uses Cases
    - React to an environmental message

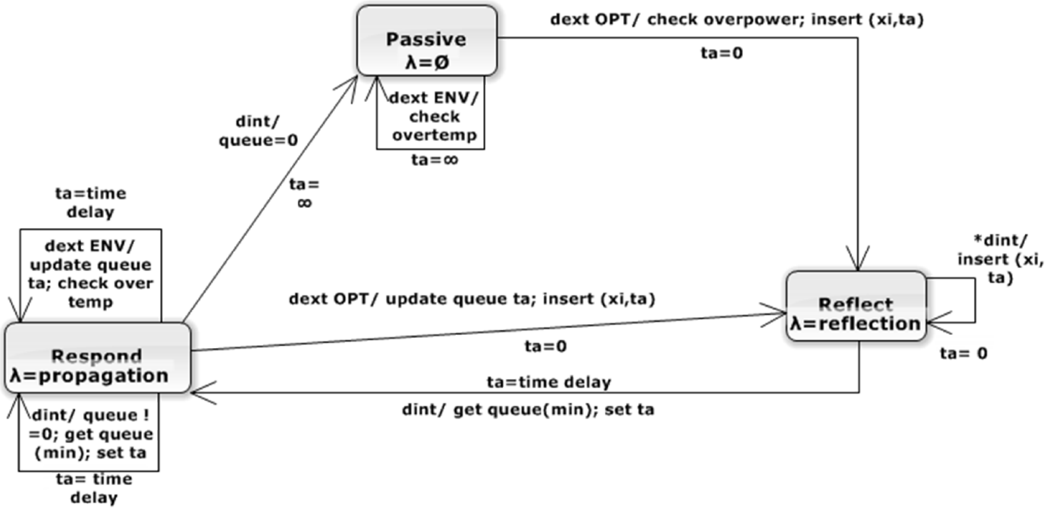
- Assumptions
    - Component has completed initialization sequence at least once
    - Reflections are not affected by component state
    - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 51. Component states.

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}


Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
dext OPT/ update queue ta; insert (xi,ta)
*dint/ insert (xi, ta)
Reflect
λ=reflection
ta= 0
Respond
λ=propagation
ta=0
ta=time delay
dint/ get queue(min); set ta
dint/ queue != 0; get queue (min); set ta
ta= time delay

* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 52. Circulator phase transition diagram.

F.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.
- Optical packet attenuated properly to the limit of accuracy.

- Optical packets propagated out the correct ports at the correct time.

F.9.3 Respond to an Environmental Packet in the Circulator

Environmental packet arrives at the circulator. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

F.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly.
- Overtemperature condition properly recognized and recorded.
- Change of state completed and recorded properly, if necessary.
- Change component function properly, if necessary.

F.10 Circulator Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change

in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Circulator Test Cases.*

		Inject Port					Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Opt3**	**Env**	Notes	opt #	**env #**
Passive	1	1	0	0	0	single	1	0
	2	0	1	0	0	single	2	0
	3	0	0	1	0	single	3	0
	4	0	0	0	1	single	3	1
	5	1	1	1	0	same time	6	1
	6	1	1	1	0	differ time	9	1
	7	1	1	1	1	same time	12	2
	8	1	1	1	1	differ time	15	3
	9	0	1	0	1	same time	16	4
	10	0	1	0	1	differ time	17	5
	11	1	0	0	1	same time	18	6
	12	1	0	0	1	differ time	19	7
	13	0	0	1	1	same time	20	8
	14	0	0	1	1	differ time	21	9
	20	2	0	0	0	same time	23	9
	21	0	2	0	0	same time	25	9
	22	0	0	2	0	same time	27	9
	23	2	2	2	0	same time	33	9
	24	2	2	2	0	differ time	39	9

	25	2	2	2	1	same time	45	10
	26	2	2	2	1	differ time	51	11
	27	0	2	0	1	same time	53	12
	28	0	2	0	1	differ time	55	13
	29	2	0	0	1	same time	57	14
	30	2	0	0	1	differ time	59	15
	31	0	0	2	1	same time	61	16
	32	0	0	2	1	differ time	63	17
totals		21	21	21	17	63		
Respond	41	2	0	0	0	single	65	17
	42	0	2	0	0	single	67	17
	43	0	0	2	0	single	69	17
	44	1	0	0	1	single	70	18
	45	2	1	1	0	same time	74	18
	46	2	1	1	0	differ time	78	18
	47	2	1	1	1	same time	82	19
	48	2	1	1	1	differ time	86	20
	49	0	2	0	1	same time	88	21
	50	0	2	0	1	differ time	90	22
	51	2	0	0	1	same time	92	23
	52	2	0	0	1	differ time	94	24
	53	0	0	2	1	same time	96	25
	54	0	0	2	1	differ time	98	26
	60	3	0	0	0	same time	101	26
	61	0	3	0	0	same time	104	26
	62	0	0	3	0	same time	107	26
	63	3	2	2	0	same time	114	26
	64	3	2	2	0	differ time	121	26
	65	3	2	2	1	same time	128	27
	66	3	2	2	1	differ time	135	28
	67	0	3	0	1	same time	138	29
	68	0	3	0	1	differ time	141	30
	69	3	0	0	1	same time	144	31
	70	3	0	0	1	differ time	147	32
	71	1	0	2	1	same time	150	33
	72	1	0	2	1	differ time	153	34
totals		38	27	25	17	90		
	TC1	1	0	0	2	single	154	36
	TC2	1	0	0	2	single	155	38
	TC3	1	0	0	2	single	156	40
	TC4	1	0	0	2	single	157	42
	TC5	1	0	0	2	single	158	44
	TC6	1	0	0	2	single	159	46

totals	6	0	0	12	6

F.11 References

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). Single mode fiber optic circulators
. Retrieved September 24, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=373

Appendix G - Optical Photodiode (Classical Detector)

G.1 Device Description

The classical detector used in Quantum Key Distribution (QKD) systems is an optical photodiode used as a photodetector. These devices rely on photogenerated charge carriers to create a response when encountering photons. A photodiode is a *p-n* junction whose current increases when it absorbs photons. The photons become absorbed in the *depletion layer* between the *p* and *n* layers, causing a current to flow between the two (Saleh & Teich, 1991). This current is linearly proportional to the amount of received photons and can be measured by standard electrical detectors.

Photodiodes used as detectors are usually of the *p-i-n* construction, where a layer of lightly doped semiconductor material is inserted between the *p* and *n* layers. The inserted layers increase the *depletion layer*, increasing the surface area for capturing photons and reducing response time See Figure 1 for examples of photodiodes (Saleh & Teich, 1991).

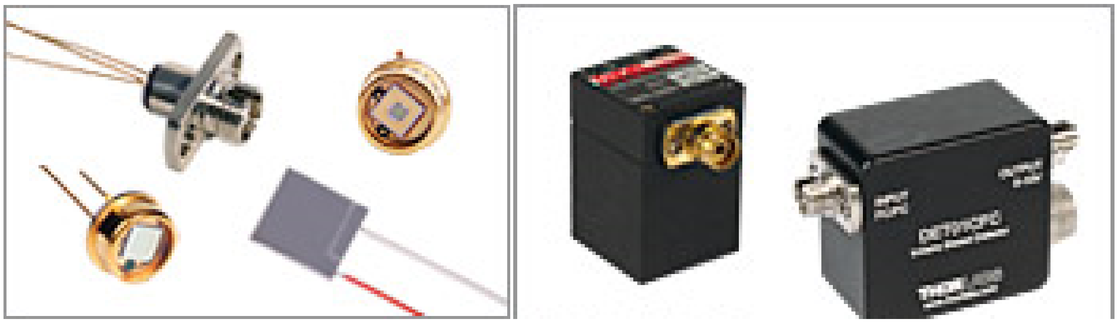


Figure 53. Example of photodiodes (ThorLabs, 2013).

Photodectectors are made from many materials that depend on the qualities desired in the device. Materials for high-speed detectors include silicon, gallium phosphide, and indium gallium arsenide. Many of these devices have a low dark current count (current caused by environment rather than the measured photons) but have a high cost for telecommunication optical fiber frequencies. See Table 1 for photodiode general characteristics.

Table 22.

Table of photodiode characteristics (ThorLabs, 2013).

Material	Dark Current	Speed	Sensitivity[a]	Cost
Silicon (Si)	Low	High Speed	400 - 1000 nm	Low
Germanium (Ge)	High	Low Speed	900 - 1600 nm	Low
Gallium Phosphide (GaP)	Low	High Speed	150 - 550 nm	Moderate
Indium Gallium Arsenide (InGaAs)	Low	High Speed	800 - 1800 nm	Moderate
Indium Arsenide Antimonide (InAsSb)	High	Low Speed	1000 - 5800 nm	High
Extended Range Indium Gallium Arsenide (InGaAs)	High	High Speed	1200 - 2600 nm	High
Mercury Cadmium Telluride (MCT, HgCdTe)	High	Low Speed	2000 - 5400 nm	High

The photodiode is a unidirectional optical component with one optical port and one electrical control port. The optical port is the input port for the optical packets with the only output being reflected signals. Optical signals arriving at the optical port are reflected back down the optical path after suffering an amount of attenuation. All received optical packets generate an electrical signal, but the signal must exceed a threshold detection level before the device signals the high-level controllers. Once an optical packet exceeds the threshold, the photodiode generates a message to its controller with a power level linearly proportionate to the incoming power level. The InGAs-type of *p*-*i*-*n* photodiode used in telecommunication wavelengths has a response time of approximately 66000ps (ThorLabs, 2013).

The photodiode is sensitive to the power of the optical signals that are received by the component. If the optical power of a pulse exceeds a defined threshold, the photodiode may become permanently damaged which changes its attenuation and output characteristics. Similarly, the photodiode is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the photodiode may become temporarily degraded or permanently damaged which changes its attenuation and output characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a "normal" operating temperature.

The first step involved with the modeling the photodiode is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the photodiode using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the photodiode. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined by the SME. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the expected behavior and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

Figure 54. Symbol for the photodiode in the QKD system architecture.

G.2 Photo-diode Conceptual Model

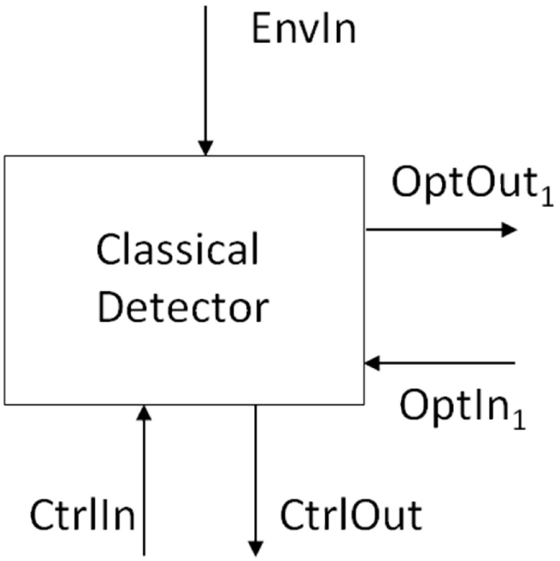


Figure 55. Photodiode conceptual model.

The conceptual model for a photodiode consists of one optical input port {OptIn$_1$ }, one optical output port {OptOut$_1$}, one environmental input port {EvnIn} and one electrical controller input port and one electrical controller output port {CtrlIn, CtrlOut}. The environmental port allows external sources to communicate changes in the operational environment to the photodiode. The electrical controller ports allow for control inputs to the controller and responses from the photodiode to the higher system functions.

In comparison to the photodiode symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. The electrical control port is not shown for clarity in Figure 2, and is also decomposed in the model into an input port and an output port. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the photodiode, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 3 will instantaneously generate a reflected emitting out of OptOut_1.

The photodiode must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation and output. External environmental messages sent to the device convey the temperature of the operational environmental so the photodiode can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the attenuation and output changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

G.3 Mathematical Model

There is no detailed mathematical description of the classical detector in Section 5.8.

G.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the classical detector (photodiode).

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.

- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Calculate the reflected power of the signal and send its output with the same port number.
- Output a control message if the optical power of the signal exceeds the threshold power level

When an environmental message arrives:

- Update the CurrrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

When a control message arrives:

- Respond to the controller with an acknowledgement message.

G.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the photodiode in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the photodiode at the same time.

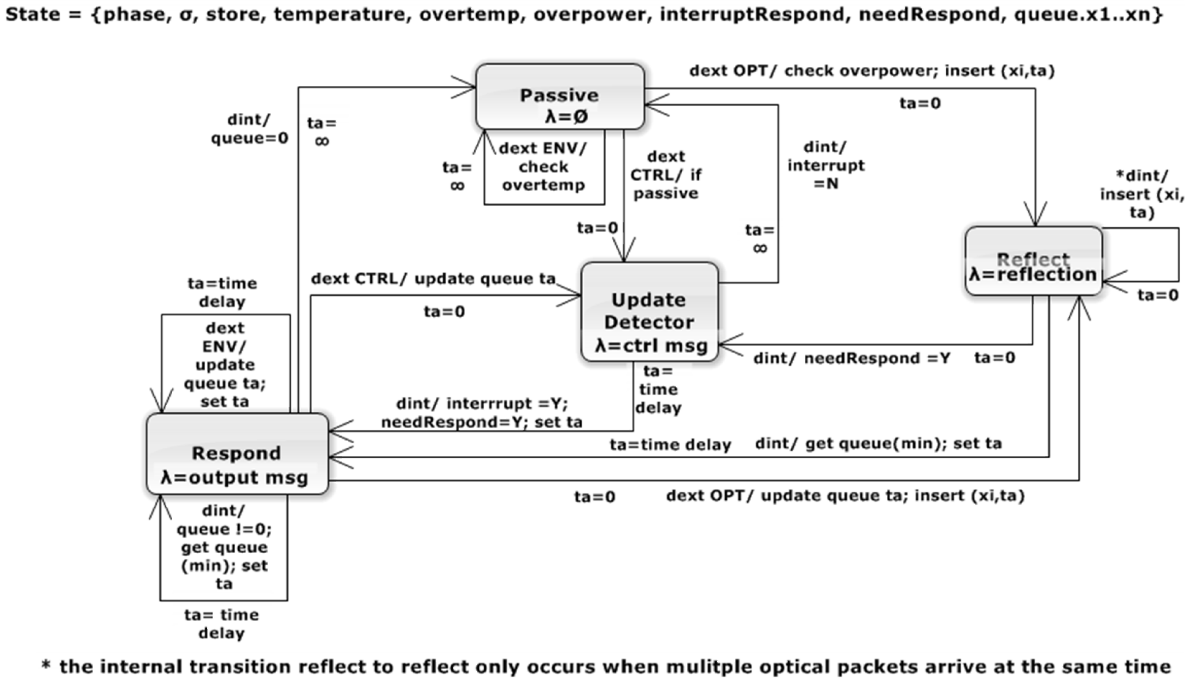


Figure 56. Photodiode phase transition diagram.

G.6 Event-Trace Diagram

This section shows various examples of packets entering the photodiode. The tables list the states the photodiode proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the photodiode, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state

- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Interrupt Respond: shows the value of the interrupt respond variable
- Need Respond: shows the value of the need respond variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

G.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 23. *Case I state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**queue (*xi*, *tp*)**	**Notes: assume tp= 6.6x10^4**
	1-packet	no env	no ext	0 ctrl								
0	s0	entry	passive	inf	null	c	n	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	entry	reflect	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	exit	reflect	6.6x10^4	x1	c	n	n	n	n	null	
0	s2	entry	respond	6.6x10^4	x1	c	n	n	n	n	null	
6.6x10^4	s2	exit	respond	inf	x1	c	n	n	n	n	null	
6.6x10^4	s3	entry	passive	inf	x1	c	n	n	n	n	null	

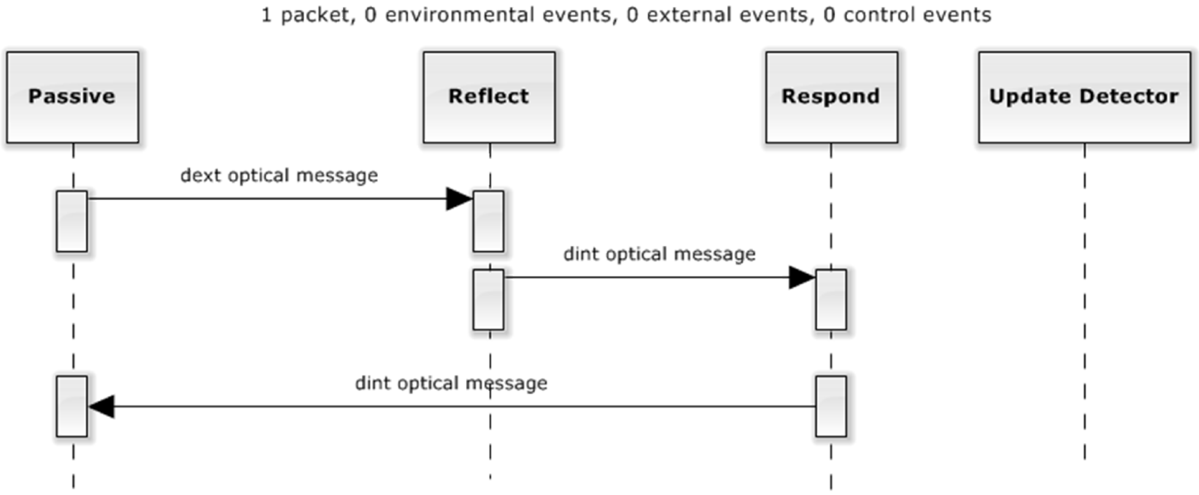


Figure 57. Case I sequence diagram.

G.6.2 Initial Passive with Single Optical Packet Arriving at Time 0 and 1 Optical Packet Arriving Time 2x10^4

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**queue (*xi*, *tp*)**	**Notes: assume tp= 6.6x10^4**
	1-packet	0 env	1 opt	0 ctrl								
0	s0	entry	passive	inf	null	c	n	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	entry	reflect	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	exit	reflect	6.6x10^4	x1	c	n	n	n	n	null	
0	s2	entry	respond	6.6x10^4	x1	c	n	n	n	n	null	
2x10^4	s2	exit	respond	0	x1	c	n	n	y	n	(x2,6.6x10^4)	dext at e= 2x10^4, 1 optical packet (x2)
2x10^4	s3	entry	reflect	0	x1	c	n	n	y	n	(x2,6.6x10^4)	
2x10^4	s3	exit	reflect	4.6x10^4	x1	c	n	n	y	n	(x2,6.6x10^4)	
2x10^4	s4	entry	respond	4.6x10^4	x1	c	n	n	y	n	(x2,6.6x10^4)	
6.6x10^4	s4	exit	respond	2x10^4	x2	c	n	n	n	n	null	
6.6x10^4	s5	entry	respond	2x10^4	x2	c	n	n	n	n	null	
8.6x10^4	s5	exit	respond	inf	x2	c	n	n	n	n	null	
8.6x10^4	s6	entry	passive	inf	x2	c	n	n	n	n	null	

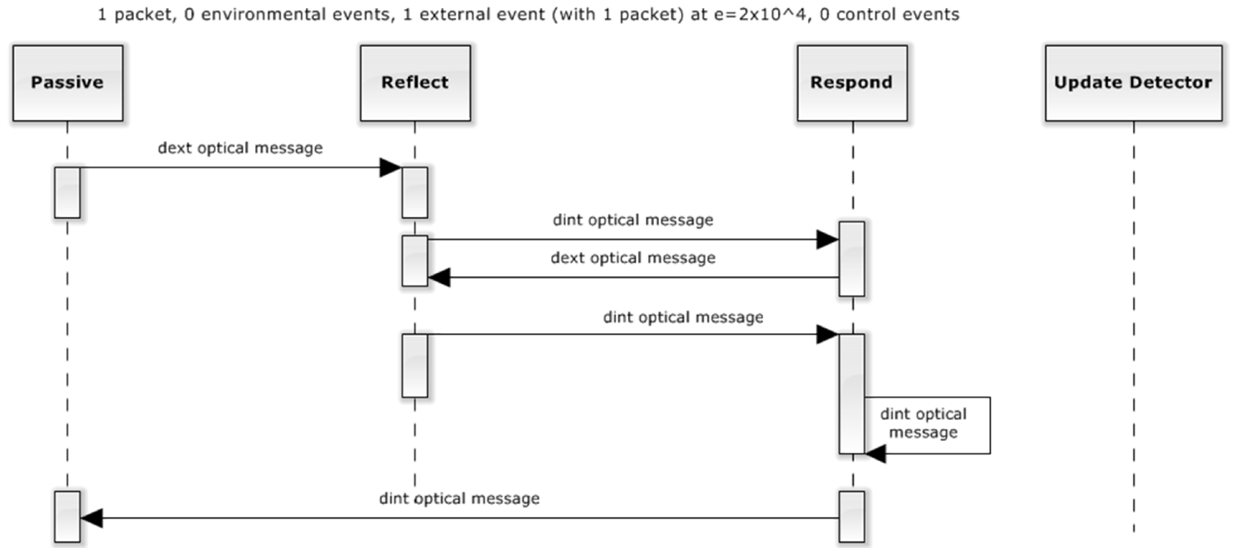


Figure 58. Case II sequence diagram.

G.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2x10^4 and Multiple Optical Packets Arriving at Time 3x10^4

Table 24. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=6.6x 10^4**
	1-packet	0 env	2 opt	0 ctrl								
0	s0	entry	passive	inf	null	c	n	n	n		null	
0	s0	exit	passive	0	null	c	n	n	n		(x1,6.6x10^4)	
0	s1	entry	reflect	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	exit	reflect	6.6x10^4	x1	c	n	n	n	n	null	
0	s2	entry	respond	6.6x10^4	x1	c	n	n	n	n	null	
2x10^4	s2	exit	respond	0	x1	c	n	n	y	n	(x2,6.6x10^4)	dext at e= 2x10^4, 1 optical packet (x2)
2x10^4	s3	entry	reflect	0	x1	c	n	n	y	n	(x2,6.6x10^4)	
2x10^4	s3	exit	reflect	4.4x10^4	x1	c	n	n	y	n	(x2,6.6x10^4)	
2x10^4	s4	entry	respond	4.4x10^4	x1	c	n	n	y	n	(x2,6.6x10^4)	
3x10^4	s4	exit	respond	0	x1	c	n	n	y	n	(x2,5.6x10^4) (x3,6.6x10^4)	dext at e= 1x10^4, 2 optical packets (x3,x4)
3x10^4	s5	entry	reflect	0	x1	c	n	n	y	n	(x2,5.6x10^4) (x3,6.6x10^4)	
3x10^4	s5	exit	reflect	0	x1	c	n	n	y	n	(x2,5.6x10^4) (x3,6.6x10^4) (x4,6.6x10^4)	
3x10^4	s6	entry	reflect	0	x1	c	n	n	y	n	(x2,5.6x10^4) (x3,6.6x10^4) (x4,6.6x10^4)	
3x10^4	s6	exit	reflect	3.3x10^4	x1	c	n	n	y	n	(x2,5.6x10^4) (x3,6.6x10^4) (x4,6.6x10^4)	

3x10^4	s7	entry	respond	3.3x10^4	x1	c	n	n	y	n	(x2,5.6x10^4) (x3,6.6x10^4) (x4,6.6x10^4)	
6.6x10^4	s7	exit	respond	2.6x10^4	x2	c	n	n	n	n	(x3,3.6x10^4) (x4,3.6x10^4)	
6.6x10^4	s8	entry	respond	2.6x10^4	x2	c	n	n	n	n	(x3,3.6x10^4) (x4,3.6x10^4)	
9.2x10^4	s8	exit	respond	1x10^4	x3	c	n	n	n	n	(x4,1x10^4)	
9.2x10^4	s9	entry	respond	1x10^4	x3	c	n	n	n	n	(x4,1x10^4)	
10.2x10^4	s9	exit	respond	0	x4	c	n	n	n	n	null	
10.2x10^4	s10	entry	respond	0	x4	c	n	n	n	n	null	
10.2x10^4	s10	exit	respond	inf	x4	c	n	n	n	n	null	
10.2x10^4	s11	entry	passive	inf	x4	c	n	n	n	n	null	

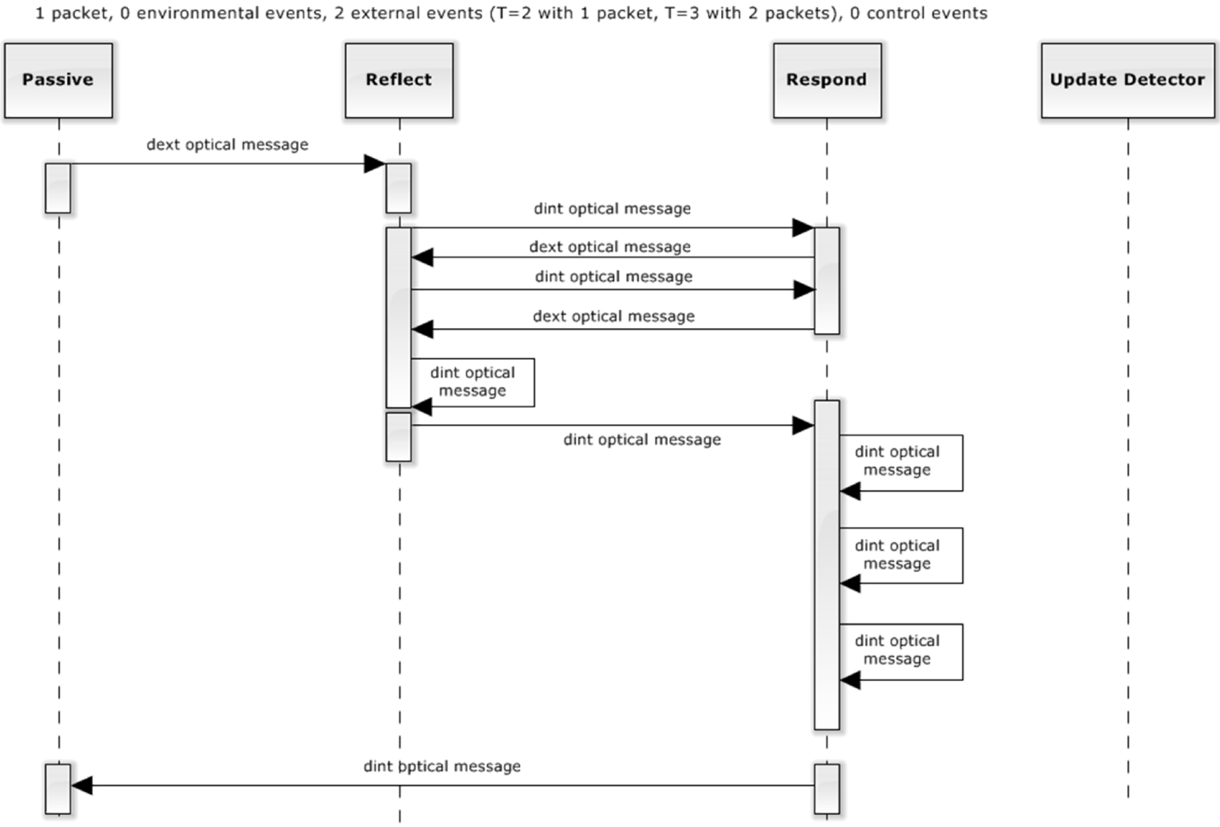
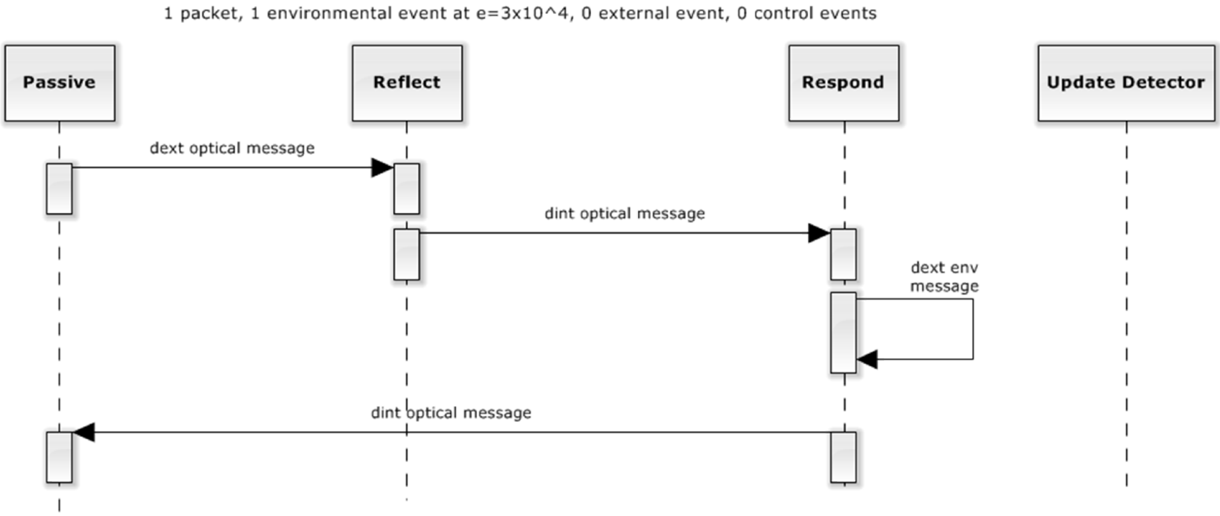


Figure 59. Case III sequence diagram.

G.6.4 *CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3x10^4*

Table 25. *Case IV state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**queue (*xi*, *tp*)**	**Notes: assume tp= 6.6x10^4**
	1-packet	1 env	0 ext	0 ctrl								
0	s0	entry	passive	inf	null	c	n	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	entry	reflect	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	exit	reflect	6.6x10^4	x1	c	n	n	n	n	(x1,6.6x10^4)	
0	s2	entry	respond	6.6x10^4	x1	c	n	n	n	n	null	ENV arrives e=3, overtemp the component
3x10^4	s2	exit	respond	3.6x10^4	x1	c	n	n	y	n	null	update temp
3x10^4	s3	entry	respond	3.6x10^4	x1	c	y	n	y	n	null	
6.6x10^4	s3	exit	respond	inf	x1	c2	y	n	n	n	null	
6.6x10^4	s4	entry	passive	inf	x1	c2	y	n	n	n	null	



G.6.5 *CASE V: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Control Packet Arriving at Time 3*

Table 26. *Case V state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**queue (xi, *tp*)**	**Notes: assume tp= 6.6x10^4**
	1 opt	1 env	0 opt	1 ctrl								
0	s0	entry	passive	inf	null	c	n	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	entry	reflect	0	null	c	n	n	n	n	(x1,6.6x10^4)	
0	s1	exit	reflect	6.6x10^4	x1	c	n	n	n	n	(x1,6.6x10^4)	
0	s2	entry	respond	6.6x10^4	x1	c	n	n	n	n	(x1,6.6x10^4)	CTRL arrives e=3x10^4
3x10^4	s2	exit	respond	0	x1	c	n	n	y	n	(x1,3.6x10^4)	
3x10^4	s3	entry	update detector	0	x1	c	n	n	y	n	(x1,3.6x10^4)	
3x10^4	s3	exit	update detector	3.6x10^4	x1	c	n	n	y	n	(x1,3.6x10^4)	
3x10^4	s4	entry	respond	3.6x10^4	x1	c	n	n	y	n	(x1,3.6x10^4)	
6.6x10^4	s4	exit	respond	0	x1	c	n	n	n	n	null	
6.6x10^4	s6	entry	passive	inf	x1	c	n	n	n	n	null	

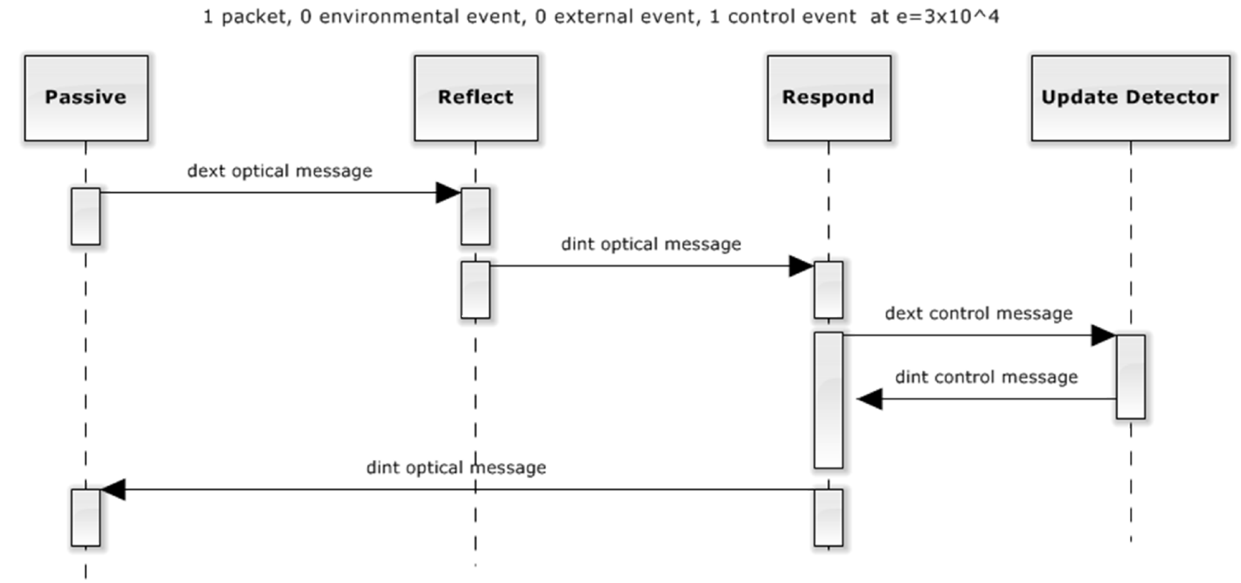


Figure 60. Case V sequence diagram.

G.7 Photodiode Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, “needRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the photodiode we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

$\boldsymbol{DEVS_{photodiode} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$

where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “reflect”, “respond”, “update detector”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
needRespond= flag variable set when both Reflect and UpdateDetector respond to inputs
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
ctrlOutput = variable that stores the output control message response
responseOutput = variable that stores the output detection message
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
ctrlMsg() = method that generates a response message to received control messages
outputMsg() = method that generates the response message to received optical packets
insert_event_q() = method that inserts the current (x_i, time delay_{i}) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay_{i}) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse

calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"OptIn_1", "EnvIn", "CtrlIn"} with
X_M = {("OptIn_1", V_{opt}), ("EnvIn", V_{env}), ("CtrlIn", V_{ctrl})} is the set of input ports and values.

OutPorts = {"OptOut_1", "CtrlOut"} with
Y_M = {("OptOut_1", $Y_{OptOut1}$), ("CtrlOut", $Y_{CtrlOut}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, needRespond, queue*}
= {{"passive", "reflect", "respond", "update detector"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x {"Y","N"} x {"Y","N"} x *V*}

External Transition Function:

δ_{ext}(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond* , *queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =
("reflect", 0, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond, queue.x1..xn*)

```
if phase = "passive" and p = "OptIn1"
for messagebag != null
current = messagebag_first()
if current.value.power > damaged.power
overpower = "Y"
insert_event_q(current)
```

```
remove_event_m(current)
queue.current = queue_first(queue)
  reflect = (queue.current.p), calcReflected(queue.current.v))
  mark_reflected(queue.current)
  interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                                        queue.x1..xn)
if phase = “respond” and p = “OptIn$_1$”
update_delay(queue)
for messagebag != null
current = messagebag_first()
if current.value.power > damaged.power
overpower =  “Y”
insert_event_q(current)
remove_event_m(current)
queue.current = queue_need_reflected()
reflect = (queue.current.p), calcReflected(queue.current.v))
  mark_reflected(queue.current)
  interruptRespond= “Y”
  timeLeftRespond = timeLeftRespond - e

(“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                                        queue.x1..xn)
if phase = “passive” and p = “EnvIn”
temperature = messagebag.temperature
if temperature > damage.temp
overtemp = “Y”

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                            needRespond,  queue.x1..xn)
if phase = “respond” and p = “EnvIn”
update_delay(queue)
timeLeftRespond = time.delay- e
temperature = messagebag.temperature
if temperature > damage.temp
overtemp = “Y”
time.delay = timeLeftRespond

(“update detector”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                              queue.x1..xn)
if phase = “passive” and p = “CtrlIn”
ctrlOutput = ctrlMsg(store)

(“update detector”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                             queue.x1..xn)
```

```
if phase = “respond” and p = “CtrlIn”
update_delay(queue)
ctrlOutput = ctrlMsg(store)
  interruptRespond= “Y”
  timeLeftRespond = timeLeftRespond - e

(phase, σ – e, store, temperature, overtemp, overpower, interruptRespond, needRespond, queue.x1..xn)
otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, queue)=
(“reflect”, 0, temperature, overtemp, overpower, interruptRespond, needRespond, queue.x1..xn))
  if phase = “reflect” and need.reflect != null
need.reflect = queue_need_reflected()
current = need.reflect
  reflect = (current.p), calcReflected(current.v))
  mark_reflected(current)


(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, needRespond, queue.x1..xn)
if phase = “reflect” and need.reflect = null
need.reflect = queue_need_reflected()
if interruptRespond = “N”
current = queue_min()
time.delay = current.time.delay
responseOutput= outputMsg(store.v, temperature, overtemp, peakpwr, overpwr)
timeLeftRespond = propagation delay
else
time.delay = timeLeftRespond

(“update detector”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond, queue.x1..xn)
if phase = “reflect” and needRespond = “Y”
ctrlOutput = ctrlMsg(store)

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, needRespond, queue.x1..xn)
if phase = “respond” and size > 0
update_delay(queue)
size= queue_size()
current = queue_min()
time.delay = current.time.delay
responseOutput= outputMsg(store.v, temperature, overtemp, peakpwr, overpwr)
```

interruptRespond= "N"

("respond", *time.delay*, *store,* *temperature*, *overtemp*, *overpower*, *interruptRespond, needRespond, queue.x*1*..xn*)
if *phase* = "update detector" and *interruptRespond* = "Y"
time.delay = *timeLeftRespond*

("respond", *time.delay*, *store*, *temperature*, *overtemp*, *overpower, overpower*, *interruptRespond, needRespond, currentAttenuation, queue.x*1*..xn*)
if *phase* = "update detector" and *interruptRespond* = "N" and *needRespond* = "Y"
current = queue_min()
time.delay = current.time.delay
responseOutput= outputMsg(*store.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
timeLeftRespond = propagation delay

("passive", ∞, *store*, *temperature*, *overtemp*, *overpower, overpower*, *interruptRespond, needRespond, queue.x*1*..xn*)

if *phase* = "update detector" and *interruptRespond* = "N"

("passive", ∞, *store,* *temperature*, *overtemp*, *overpower, interruptRespond, needRespond, queue.x*1*..xn*)
if *phase* = "respond" and *size* = 0
size= queue_size()

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:
λ(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond, queue*) = (*reflect.p, reflect.v*)
if phase = "reflect"

(Output_1, *responseOutput*)
if phase = "respond"

("CtrlOut", *ctrlOutput*)
if phase = "update detector"

∅ (null output)
otherwise;

Time advance Function:

ta(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond, queue*) = *σ*;

G.8 Mathematical model

```
// intrinsic parameters
threshold = par("threshold");                        // e field threshold (V/m)
conversionFactor = par("conversionFactor");          // conversion factor
generateReflections = par("generateReflections");
insertionLoss = par("insertionLoss");                // insertion loss (dB)
returnLoss = par("returnLoss");                      // return loss (dB)
degradedAttenuation = par("degradedAttenuation");    // degraded attenuation (0.0-1.0)
damagedAttenuation = par("damagedAttenuation");      // damaged attenuation (0.0-1.0)
tempDegradeThreshold = par("tempDegradeThreshold");  // temperature degrade threshold (C)
tempDamageThreshold = par("tempDamageThreshold");    // temperature damage threshold (C)
powerDegradeThreshold = par("powerDegradeThreshold"); // power degrade threshold (Watts)
powerDamageThreshold = par("powerDamageThreshold");  // power damage threshold (Watts)
propagationDelay = par("propagationDelay");          // propagation delay (sec)
displayInputPulse = par("displayInputPulse");        // display input pulse parameters
computeInputPulsePower = par("computeInputPulsePower"); // compute input pulse power


// extrinsic parameters
temperatureMean = par("temperatureMean");            // temperature mean (C)
temperatureStdDev = par("temperatureStdDev");        // temperature standard deviation (C)
```

Generate a reflection:

```
outAmplitude = amplitude * std::sqrt((std::pow(10.0, (-1.0*returnLoss/10.0))));
outGlobalPhase = globalPhaseRange(globalPhase+PI);
outOrientation = orientationRange(orientation);
outEllipticity = ellipticityRange(ellipticity);
outCentralFreq = centralFreq;
```

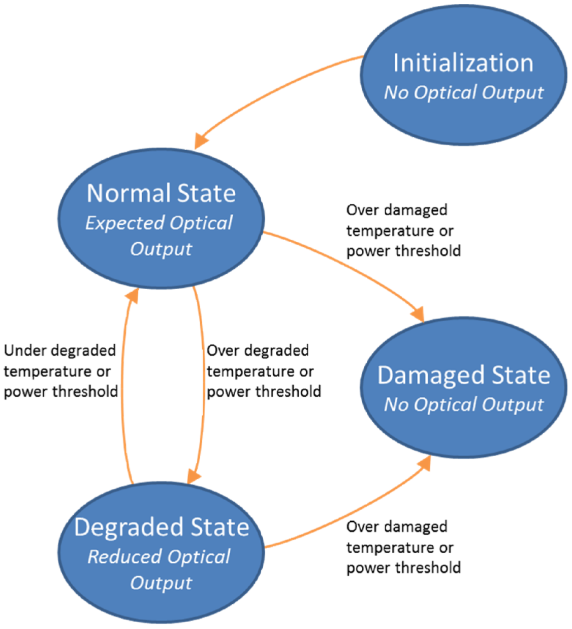
G.9 Component Use Case

G.9.1 Respond to an Optical Packet in the Classical Detector (CD)

Optical packet arrives at the CD. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the CD. Records overpower condition, if applicable. Remove the optical packet from the queue and create a control output signal based on the input signal and the current component state. Send the control output signal out of the component control port.

- Identified Alternative Uses Cases
  - Respond to a control message
  - React to an environmental message

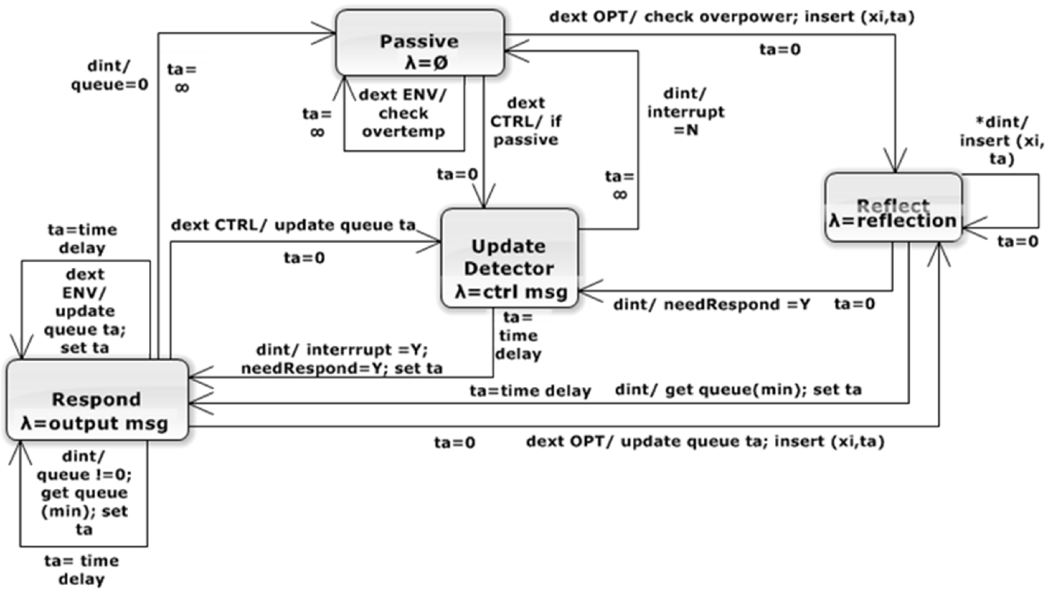
- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 61. Component states.

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, queue.x1..xn}



* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 62. Photodiode phase transition diagram.

G.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.

- Control message created and sent out the correct port at the correct time.

G.9.3 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.
- Optical packet attenuated properly to the limit of accuracy.
- Optical packet propagated out the correct port at the correct time.

G.9.4 Respond to an Environmental Packet in the Classical Detector (CD)

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - o None

G.9.5 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

G.9.6 Respond to a Control Message in the Classical Detector (CD)

Control Message arrives at the component. Component decodes message properly. Records change in condition or state, if applicable. Change component function if in degraded or damaged state or by change in component condition, if necessary.

- Assumptions
    - o Component has completed initialization sequence at least once

G.9.7 Respond to Control Message End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

G.10 Optical Photo-diode Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical

SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 6. *Optical Photo-diode Test Cases*

		Inject Port				Running Totals		
						opt	**env**	**ctrl**
Phase	**Case**	**Opt1**	**Ctrl**	**Env**	Notes	#	**#**	**#**
Passive	1	1	0	0	single	1	0	0
	2	0	1	0	single	1	0	1
	3	0	0	1	single	1	1	1
	4	1	1	0	same time	2	1	2
	5	1	1	0	differ time	3	1	3
	6	1	1	1	same time	4	2	4
	7	1	1	1	differ time	5	3	5
	8	0	1	1	same time	5	4	6
	9	0	1	1	differ time	5	5	7
	10	1	0	1	same time	6	6	7
	11	1	0	1	differ time	7	7	7
	20	2	0	0	same time	9	7	7
	21	0	1	0	same time	9	7	8
	22	2	1	0	same time	11	7	9
	23	2	1	0	differ time	13	7	10
	24	2	1	1	same time	15	8	11
	25	2	1	1	differ time	17	9	12
	26	0	1	1	same time	17	10	13
	27	0	1	1	differ time	17	11	14
	28	2	0	1	same time	19	12	14
	29	2	0	1	differ time	21	13	14
totals		21	14	13				
Respond	41	2	0	0	single	23	13	14
	42	1	1	0	single	24	13	15
	43	1	0	1	single	25	14	15
	44	2	1	0	same time	27	14	16
	45	2	1	0	differ time	29	14	17

46	2	1	1	same time	31	15	18
47	2	1	1	differ time	33	16	19
48	1	1	1	same time	34	17	20
49	1	1	1	differ time	35	18	21
50	2	0	1	same time	37	19	21
51	2	0	1	differ time	39	20	21
60	3	0	0	same time	42	20	21
61	1	1	0	same time	43	20	22
62	3	1	0	same time	46	20	23
63	3	1	0	differ time	49	20	24
64	3	1	1	same time	52	21	25
65	3	1	1	differ time	55	22	26
66	1	1	1	same time	56	23	27
67	1	1	1	differ time	57	24	28
68	3	0	1	same time	60	25	28
69	3	0	1	differ time	63	26	28
totals	42	14	13				
TC1	1	1	2	single	64	28	29
TC2	1	1	2	single	65	30	30
TC3	1	1	2	single	66	32	31
TC4	1	1	2	single	67	34	32
TC5	1	1	2	single	68	36	33
TC6	1	1	2	single	69	38	34
TC7	1	0	2	single	70	40	34
totals	7	6	14				

Notes:

23 - INIT control message sent; OPT1 & Ctrl - differ time - Passive

24 - INIT control message sent - OPT1 & Ctrl - same time - Passive

63 - INIT control message sent - OPT1 & Ctrl - same time - Respond

66 - INIT control message sent - Ctrl & ENV - same time - Respond

G.11 References

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). Photodiode tutorial. Retrieved October 15, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=2822

Appendix H - Electronically Controlled Variable Optical Attenuator (EVOA)

H.1 Device Description

The EVOA is used to attenuate the power of optical signals by a variable amount, usually expressed in decibels (dBs). These devices usually have some form of blocking material such as an opaque slab or a window that is tilted in the path of the light. This blocking material is connected to an electric motor that is controlled by the higher system functions, allowing for a variable amount of light to exit the device. Broadband versions of the device may use a tilting window that the light passes through rather than an opaque block. See Figure 1 for an example of the internals of a VOA and Figure 2 for an example of an EVOA.

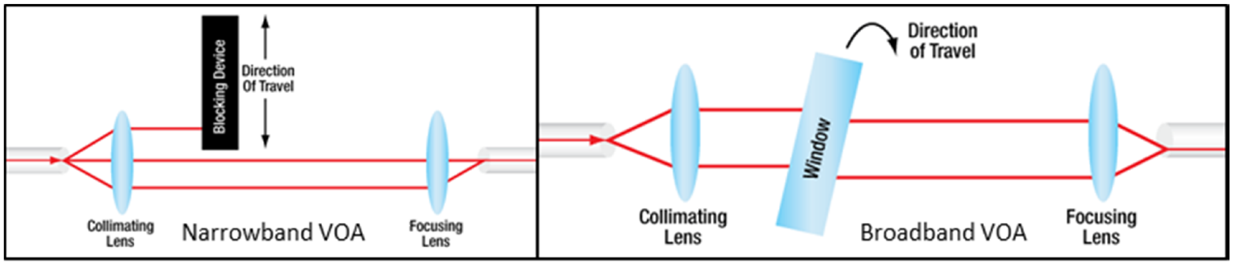


Figure 63. Example of the internals of a VOA (ThorLabs, 2013).

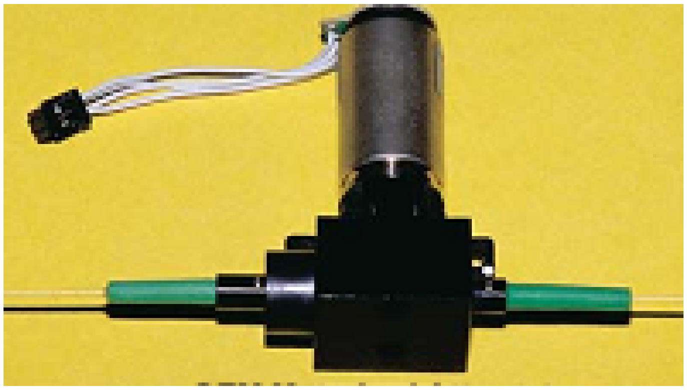


Figure 64. View of an EVOA (OZOptics, 2013)

The EVOA is a bidirectional optical component with two optical ports. Optical signals arriving at one of the ports is attenuated and propagated to the other port after a defined propagation delay. The EVOA is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the EVOA may become permanently damaged which changes its attenuation characteristics. Similarly, the EVOA is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the EVOA may become temporarily degraded or permanently damaged which changes its attenuation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the EVOA is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the EVOA. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the EVOA to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the EVOA using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the EVOA. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica

worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

Figure 65. Symbol for the EVOA in the QKD system architecture.

H.2 EVOA Conceptual Model

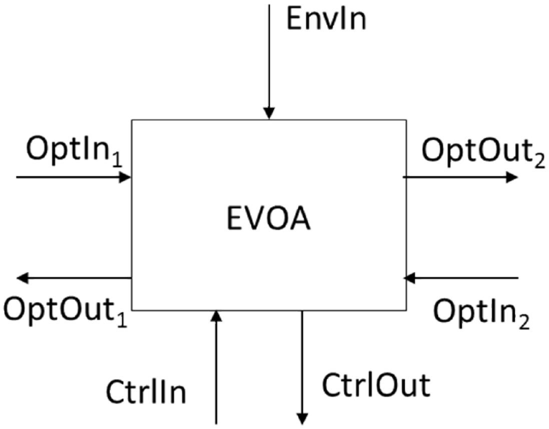


Figure 66. EVOA conceptual model.

The conceptual model for an EVOA consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, one environmental input port {EvnIn} and one electrical controller input port and one electrical controller output port {CtrlIn, CtrlOut}. The environmental port allows external sources to communicate changes in the operational environment to the EVOA. The electrical controller ports allow for control inputs to the controller and responses from the EVOA to the higher system functions.

In comparison to the EVOA symbol used in the QKD simulation architecture shown in Figure 3, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. The electrical control port is not shown for clarity in Figure 2, and is also decomposed in the model into an input port and an output port. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the EVOA, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 4 will instantaneously generate a reflected emitting out of OptOut_1.

The EVOA must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the EVOA can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

H.3 Mathematical Model

For a detailed mathematical description of the EVOA, refer to Section 6.8 which contains the Mathematica worksheet provided by the optical physics SME.

H.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the EVOA.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue
- Immediately calculate the reflected power of the signal and send its output with the same port number.

- Remove the packet from the queue; calculate the attenuated output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

When a control message arrives:

- Increase or decrease the attenuation per the control message as a function of time and the attenuation rate of change.
- Respond to controller if the maximum or minimum attenuation of the EVOA is reached and ensure that these values are not exceeded.

H.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the EVOA in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the EVOA at the same time.

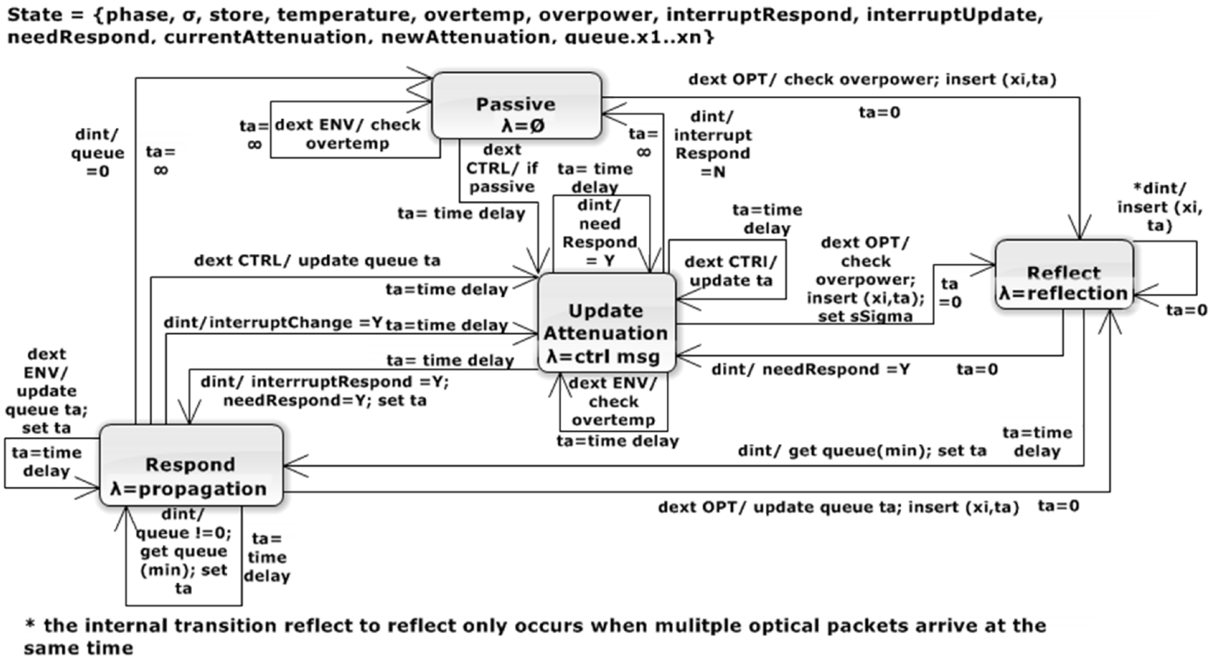

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate, needRespond, currentAttenuation, newAttenuation, queue.x1..xn}
Passive λ=Ø
Update Attenuation λ=ctrl msg
Reflect λ=reflection
Respond λ=propagation
dext OPT/ check overpower; insert (xi,ta)
ta=0
dint/ queue =0
ta= ∞
ta= ∞
dext ENV/ check overtemp
ta= ∞
dint/ interrupt Respond =N
dext CTRL/ if passive
ta= time delay
ta= time delay
dint/ need Respond = Y
ta=time delay
dext CTRl/ update ta
dext OPT/ check overpower; insert (xi,ta); set sSigma
ta =0
*dint/ insert (xi, ta)
ta=0
dext CTRL/ update queue ta
ta=time delay
dint/interruptChange =Y
ta=time delay
ta= time delay
dint/ interrruptRespond =Y; needRespond=Y; set ta
dint/ needRespond =Y
ta=0
dext ENV/ check overtemp
ta=time delay
dext ENV/ update queue ta; set ta
ta=time delay
dint/ get queue(min); set ta
ta=time delay
dext OPT/ update queue ta; insert (xi,ta)
ta=0
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 67. EVOA phase transition diagram.

H.6 Event-Trace Diagram

This section shows various examples of packets entering the EVOA. The tables list the states the EVOA proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the EVOA, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable

- Queue: contents of the queue for that state
- Notes: any notes for that state

H.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 27. *Case I state list*.

time	state	entry/exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	interrupt update	need respond	current atten	new atten	queue (*xi*, *tp*)	Notes: assume tp= 5
0	s0	entry	passive	inf	null	c	n	n	n	n	n	dB	dB	null	
0	s0	exit	passive	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	n	dB	dB	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	n	dB	dB	null	
5	s2	exit	respond	inf	x1	c	n	n	n	n	n	dB	dB	null	
5	s3	entry	passive	inf	x1	c	n	n	n	n	n	dB	dB	null	

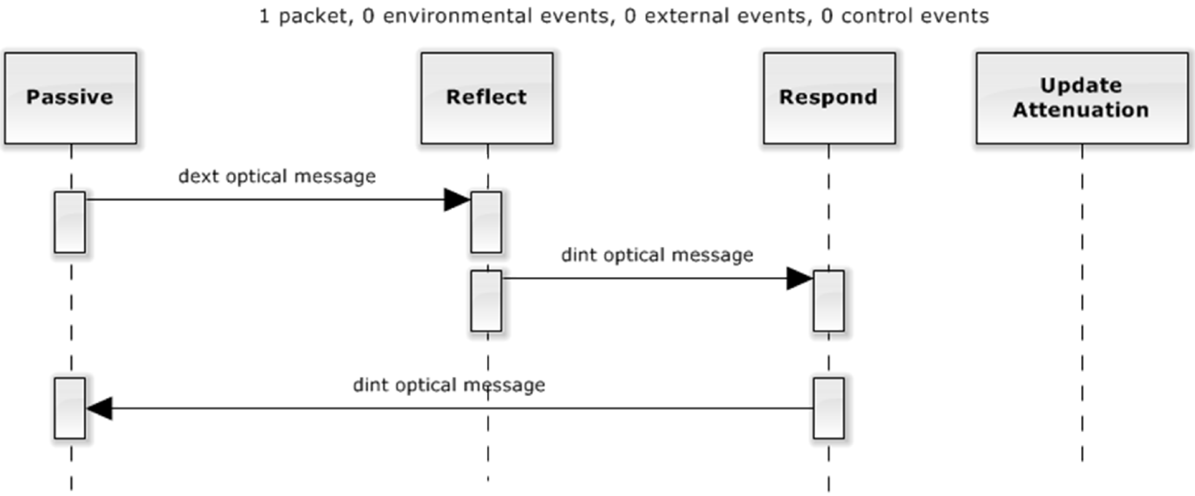


Figure 68. Case I sequence diagram.

H.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 28. *Case II state list*.

time	state	entry/exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	interrupt update	need respond	current atten	current atten	queue (*xi*, *tp*)	Notes: assume tp= 5
0	s0	entry	passive	inf	null	c	n	n	n	n	n	dB	dB	null	

0	s0	exit	passive	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	n	dB	dB	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	n	dB	dB	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	n	dB	dB	(x2,5)	dext at e=2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	n	dB	dB	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	n	dB	dB	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	n	dB	dB	(x2,5)	
5	s4	exit	respond	2	x2	c	n	n	n	n	n	dB	dB	null	
5	s5	entry	respond	2	x2	c	n	n	n	n	n	dB	dB	null	
7	s5	exit	respond	inf	x2	c	n	n	n	n	n	dB	dB	null	
7	s6	entry	passive	inf	x2	c	n	n	n	n	n	dB	dB	null	

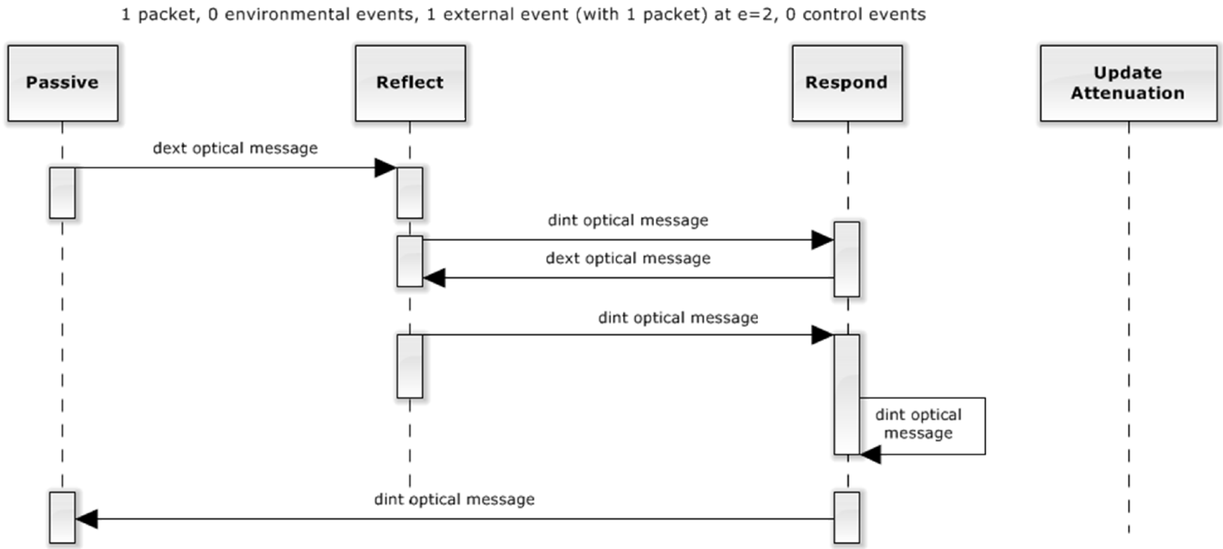


Figure 69. Case II sequence diagram.

H.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 29. *Case III state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	interrupt update	need respond	current atten	current atten	queue (x*i*, *tp*)	Notes: assume tp=5

0	s0	entry	passive	inf	null	c	n	n	n	n	n	dB	dB	null	
0	s0	exit	passive	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	n	dB	dB	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	n	dB	dB	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	n	dB	dB	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	n	dB	dB	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	n	dB	dB	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	n	dB	dB	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	n	n	dB	dB	(x2,4) (x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	n	n	dB	dB	(x2,4) (x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	n	n	dB	dB	(x2,4) (x3,5) (x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	n	n	dB	dB	(x2,4) (x3,5) (x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	n	n	dB	dB	(x2,4) (x3,5) (x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	n	n	dB	dB	(x2,4) (x3,5) (x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	n	n	dB	dB	(x3,2) (x4,2)	
5	s8	entry	respond	2	x2	c	n	n	n	n	n	dB	dB	(x3,2) (x4,2)	
7	s8	exit	respond	1	x3	c	n	n	n	n	n	dB	dB	(x4,0)	
7	s9	entry	respond	1	x3	c	n	n	n	n	n	dB	dB	(x4,0)	
8	s9	exit	respond	0	x4	c	n	n	n	n	n	dB	dB	null	
8	s10	entry	respond	0	x4	c	n	n	n	n	n	dB	dB	null	
8	s10	exit	respond	inf	x4	c	n	n	n	n	n	dB	dB	null	
8	s11	entry	passive	inf	x4	c	n	n	n	n	n	dB	dB	null	

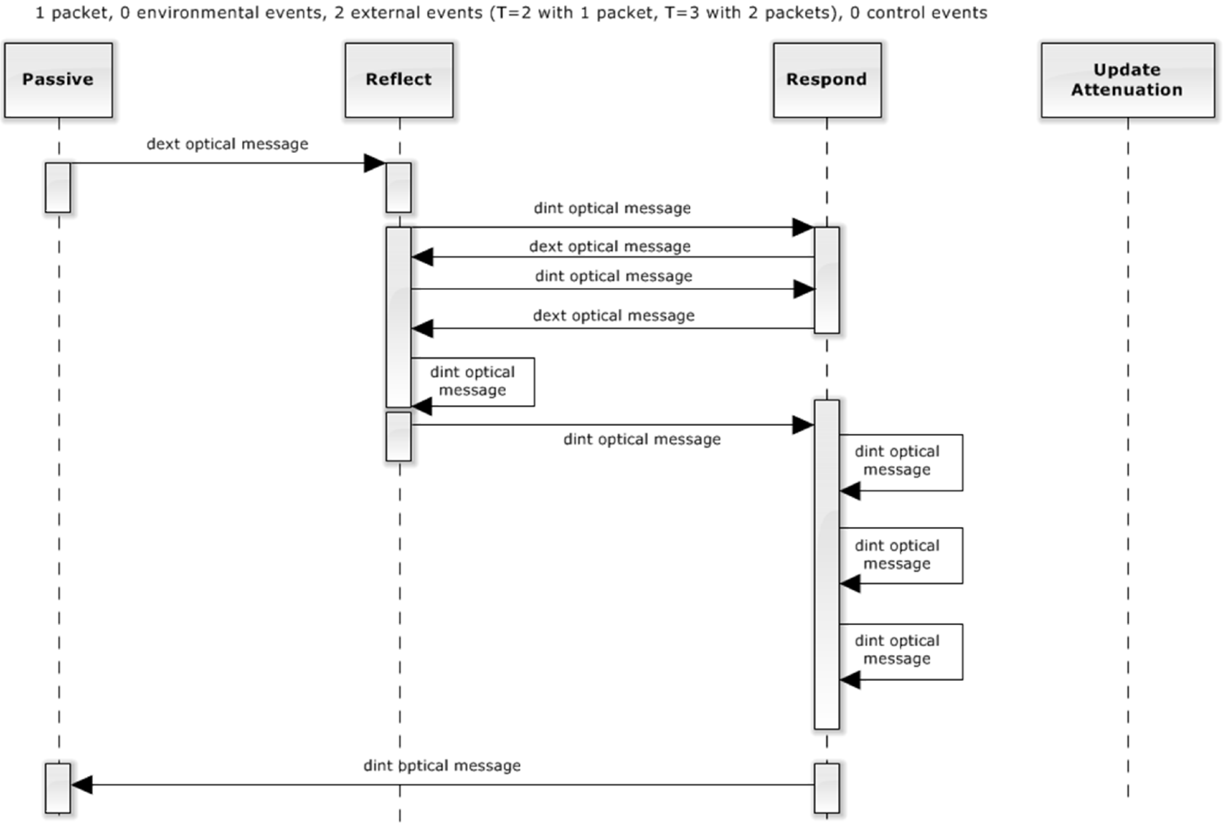


Figure 70. Case III sequence diagram.

H.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 30. *Case IV state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	interrupt update	need respond	current atten	current atten	queue (x*i*, *tp*)	Notes: assume tp= 5
0	s0	entry	passive	inf	null	c	n	n	n	n	n	dB	dB	null	
0	s0	exit	passive	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	n	dB	dB	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	y	n	n	dB	dB	null	update temp
3	s3	entry	respond	2	x1	c	y	n	y	n	n	dB	dB	null	
5	s3	exit	respond	inf	x1	c2	y	n	n	n	n	dB	dB	null	

5	s4	entry	passive	inf	x1	c2	y	n	n	n	n	dB	dB	null	

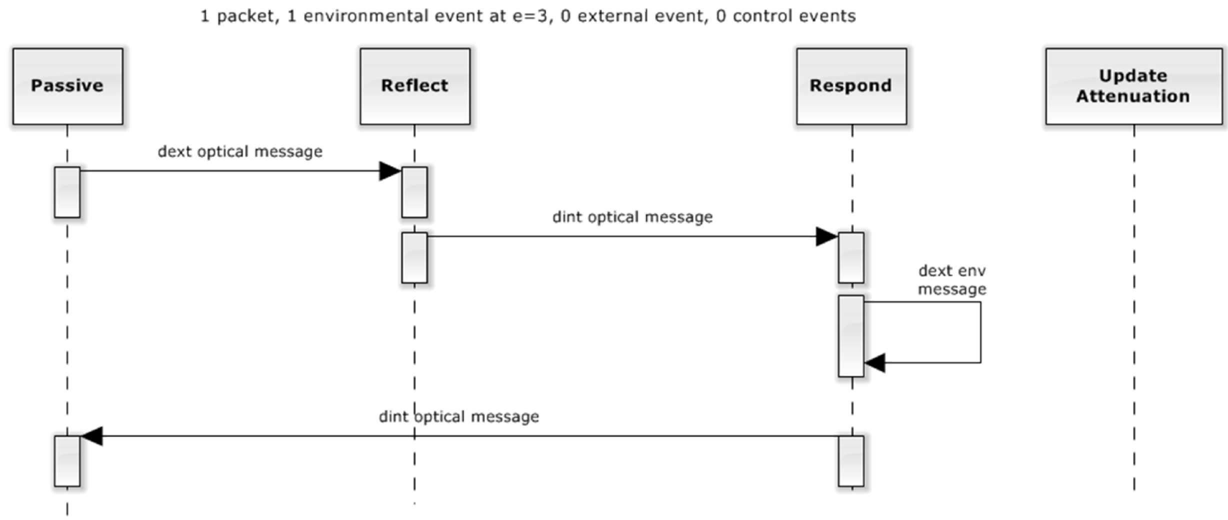


Figure 71. Case IV sequence diagram.

H.6.5 CASE V: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Control Packet Arriving at Time 3

Table 31. *Case V state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	interrupt update	need respond	current atten	current atten	queue (x*i*, *tp*)	Notes: assume tp= 5
0	s0	entry	passive	inf	null	c	n	n	n	n	n	dB	dB	null	
0	s0	exit	passive	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	n	dB	dB	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	n	dB	dB	(x1,5)	CTRL arrives e=3
3	s2	exit	respond	0	x1	c	n	n	y	n	n	dB	dB	(x1,2)	
3	s3	entry	update atten	0	x1	c	n	n	y	n	n	dB	dB	(x1,2)	
3	s3	exit	update atten	2	x1	c	n	n	y	n	n	dB	dB	(x1,2)	
3	s4	entry	respond	2	x1	c	n	n	y	n	n	dB	dB	(x1,2)	
5	s4	exit	respond	0	x1	c	n	n	n	n	n	dB	dB	null	
5	s6	entry	passive	inf	x1	c	n	n	n	n	n	dB	dB	null	

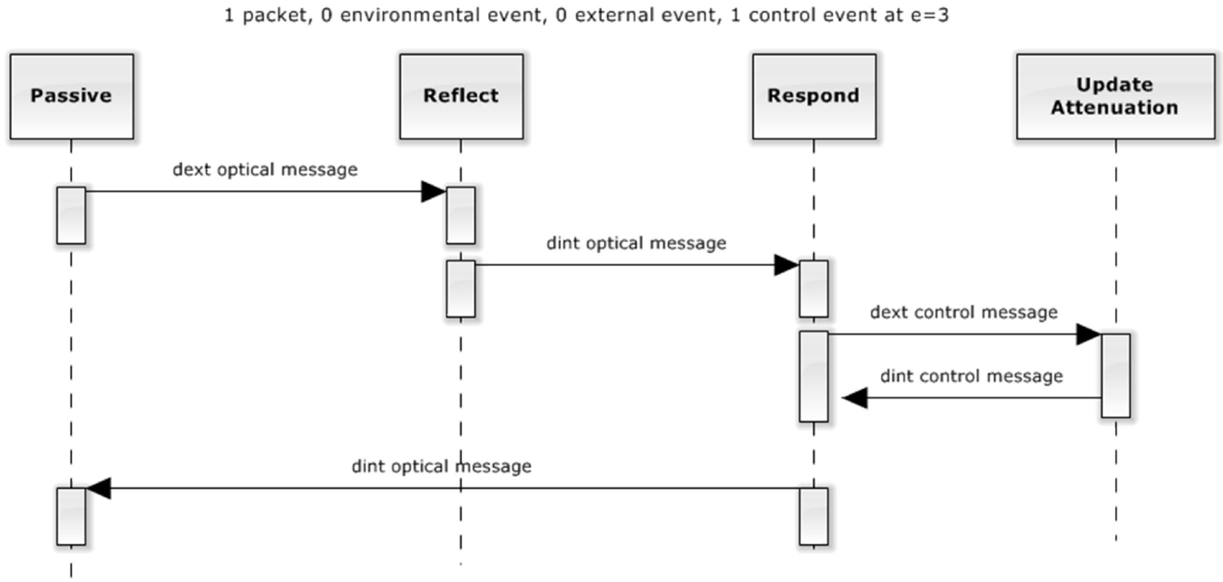


Figure 72. Case V sequence diagram.

H.7 EVOA Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- Assume that only one control packet will arrive at any given time, due to the small time scales involved and the length of time necessary for attenuation changes.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”,”interruptRespond”, “interruptUpdate”, “needRespond”, “currentAttenuation”, “newAttenuation”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred

"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level
"overpower" = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the EVOA we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \text{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \to S$ is the external state transition function;
$\delta_{int} = S \to S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \to S$ is the confluent transition function;
$\lambda = S \to Y^b$ is the output function;
$ta = S \to R_0^+ \cup \infty$ or $S \to R_{0^+ \to \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS$_{EVOA}$* = (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {"passive", "reflect", "respond", "update attenuation"}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
interruptUpdate = flag variable set when UpdateAttenuation phase is interrupted by an external event
needRespond= flag variable set when both Reflect and UpdateAttenuation respond to inputs
currentAttenuation = current attenuation of the EVOA
newAttenuation = attenuation the EVOA is changing to after receiving a change message
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current x input value(s) (p,v)
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter

current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
ctrlOutput = variable that stores the output control message response
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
attenuationMin = minimum selectable attenuation
attenuationMax = maximum selectable attenuation
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
ctrlMsg() = method that generates a response message to received control messages
outputMsg() = method that generates the response message to received optical packets
insert_event_q() = method that inserts the current (x_i, time delay_i) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay_i) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr, currentAttenuation, newAttenuation*)
changeAttenuation() = method that changes current attenuation of the EVOA
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet

q.v = pointer to a value in the queue
$q.v_{min}$ = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"OptIn_1", "OptIn_2", "EnvIn", "CtrlIn"} with

X_M = {(“OptIn$_1$”, V_{opt}), (“OptIn$_2$”, V_{opt}), (“EnvIn”, V_{env}), (“CtrlIn”, V_{ctrl})} is the set of input ports and values.

OutPorts = {“OptOut$_1$”, “OptOut$_2$”, “CtrlOut”} with
Y_M = {(“OptOut$_1$”, $Y_{OptOut1}$), (“OptOut$_2$”, $Y_{OptOut2}$), (“CtrlOut”, $Y_{CtrlOut}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, interruptUpdate, needRespond, currentAttenuation, newAttenuation, queue*} = {{“passive”, “reflect”, “respond”, “update attenuation”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x {“Y”,”N”} x {“Y”,”N”} x *V* x *V* x *V*}

External Transition Function:

```
δext(phase, σ, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
        needRespond , currentAttenuation, newAttenuation, queue, e, ((pi,vi),…. (pn,vn))) =
("reflect", 0, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
                              needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = "passive" and p ∈ {"OptIn1", "OptIn2"}
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower = "Y"
      insert_event_q(current)
      remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = "N"

("reflect", 0, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
                              needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = "update attenuation" and p ∈ {"OptIn1", "OptIn2"}
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower = "Y"
      insert_event_q(current)
      remove_event_m(current)
    queue.current = queue.first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
   if currentAttenuation != newAttenuation
```

```
    currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr,
currentAttenuation, newAttenuation)
   timeLeftUpdate = timeLeftUpdate- e

 (“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
                                 needRespond, currentAttenuation, newAttenuation queue.x1..xn)
   if phase = “respond” and $p \in \{“OptIn_1”, “OptIn_2”\}$
     update_delay(queue)
     for messagebag != null
       current = messagebag_first()
       if current.value.power > damaged.power
         overpower =  “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_need_reflected()
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond= “Y”
    if currentAttenuation != newAttenuation
      currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr,
    timeLeftRespond = timeLeftRespond - e

(“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
                                 needRespond, currentAttenuation, newAttenuation queue.x1..xn)
    if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                  interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
    if phase = “respond” and p = “EnvIn”
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”
     if current.Attenuation != newAttenuation
       currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr,
     time.delay = timeLeftRespond

 (“update attenuation”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                  interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
    if phase = “update attenuation” and p = “EnvIn”
     update_delay(queue)
     temperature = messagebag.temperature
    if temperature > damage.temp
     overtemp = “Y”
    if currentAttenuation != newAttenuation
     currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
    timeLeftUpdate = time.delay- e
    time.delay = timeLeftUpdate

(“update attenuation”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
            interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)

  if phase = “passive” and p = “CtrlIn”
    ctrlOutput = ctrlMsg(store)
    currentAttenuation = changeAttenuation(store)
    if currentAttenuation < attenuationMin
      currentAttenuation = attenuationMin
    if currentAttenuation < attenuationMax
      currentAttenuation = attenuationMax

(“update attenuation”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
            interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “respond” and p = “CtrlIn”
    update_delay(queue)
    ctrlOutput = ctrlMsg(store)
    if currentAttenuation != newAttenuation
     currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr,
    currentAttenuation = changeAttenuation(store)
    if currentAttenuation < attenuationMin
      currentAttenuation = attenuationMin
    else
      if currentAttenuation < attenuationMax
        currentAttenuation = attenuationMax
    interruptRespond= “Y”
    timeLeftRespond = timeLeftRespond - e

(“update attenuation”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
            interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “update attenuation” and p = “CtrlIn”
    update_delay(queue)
    ctrlOutput = ctrlMsg(store)
    if currentAttenuation != newAttenuation
     currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
    currentAttenuation = changeAttenuation(store)
    if currentAttenuation < attenuationMin
      currentAttenuation = attenuationMin
    else
      if currentAttenuation < attenuationMax
        currentAttenuation = attenuationMax
    interruptUpdate= “Y”
    timeLeftUpdate= timeLeftUpdate - e
```

```
(phase, $\sigma - e$, $\infty$, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
                                    needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
                                    needRespond, currentAttenuation, newAttenuation queue)=
(“reflect”, 0, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
                                    needRespond, currentAttenuation, newAttenuation queue.x1..xn))
  if phase = “reflect” and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                 interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “reflect” and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = “N”
      current = queue.min()
      time.delay = current.time.delay
      if InPort = “$OptIn_1$”
       outputPulse = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr,
currentAttenuation, newAttenuation)
       outputPort = “$OptOut_2$”
      if InPort = “$OptIn_2$”
       outputPulse = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr,
currentAttenuation, newAttenuation)
       outputPort = “$OptOut_1$”
     timeLeftRespond = propagation delay
   else
    time.delay = timeLeftRespond

 (“update attenuation”, 0, store, temperature, overtemp, overpower, interruptRespond,
                 interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “reflect” and needRespond = “Y”
   ctrlOutput = ctrlMsg(store)
```

```
(“update attenuation”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
            interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “respond”  and (currentAttenation != newAttenuation)
    if currentAttenuation != newAttenuation
      currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
   time.delay = timeLeftUpdate

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
            interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “respond” and size > 0
    update_delay(queue)
    size= queue_size()
    current = queue_min()
    time.delay = current.time.delay
    if currentAttenuation != newAttenuation
      currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
    if InPort = “OptIn$_1$”
      outputPulse = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
      outputPort = “OptOut$_2$”
    if InPort = “OptIn$_2$”
      outputPulse = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr)
      outputPort = “OptOut$_1$”
   interruptRespond= “N”
   if currentAttenuation != newAttenuation
   currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)

(“passive”, ∞, store, temperature, overtemp, overpower, overpower, interruptRespond,
            interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “update attenuation” and interruptRespond = ‘‘N”
    interruptUpdate = “N”
    needRespond = “N”
    interruptRespond = “N”

(“respond”, time.delay, store, temperature, overtemp, overpower, overpower, interruptRespond,
            interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “update attenuation” and interruptRespond = “Y”
    if currentAttenuation != newAttenuation
      currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
    time.delay = timeLeftRespond
```

```
(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
              interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “update attenuation” and interruptRespond = “N” and needRespond = “Y”
    current = queue.min()
    time.delay = current.time.delay
    if currentAttenuation != newAttenuation
     currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
    if InPort = “OptIn$_1$”
      outputPulse = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
      outputPort = “OptOut$_2$”
    if InPort = “OptIn$_2$”
      outputPulse = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr, currentAttenuation, newAttenuation)
      outputPort = “OptOut$_1$”
 timeLeftRespond = propagation delay

(“update attenuation”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
              interruptUpdate, needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “update attenuation” and interruptUpdate = “Y” and needRespond = “N”
    if current.Attenuation != newAttenuation
      currentAttenuation = calcAtten(store.v, temperature, overtemp, peakpwr, overpwr,
   time.delay = timeLeftUpdate

(“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
              needRespond, currentAttenuation, newAttenuation queue.x1..xn)
  if phase = “respond” and size  = 0 and (currentAttenation != newAttenuation)
    size= queue_size()
    interruptUpdate = “N”
    needRespond = “N”
    interruptRespond = “N”
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

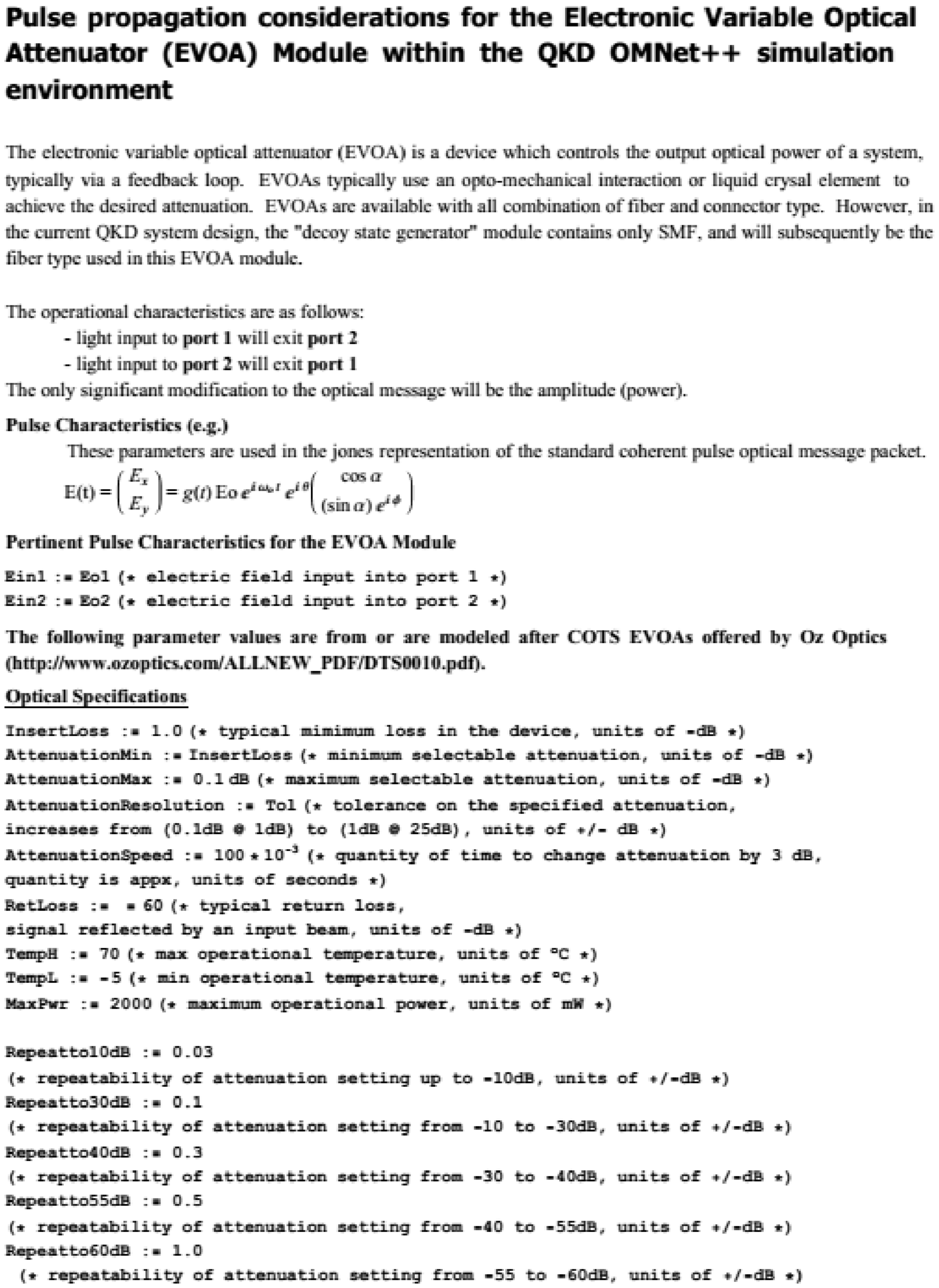
```
λ(phase, σ, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate,
              needRespond, currentAttenuation, newAttenuation queue) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (outputPort, outputPulse)
    if phase = “respond”

  (“CtrlOut”, ctrlOutput)
```

if phase = “update attenuation”

∅ (null output)
otherwise;

Time advance Function:

ta(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, interruptUpdate, needRespond, currentAttenuation, newAttenuation queue*) = *σ*;



Pulse propagation considerations for the Electronic Variable Optical Attenuator (EVOA) Module within the QKD OMNet++ simulation environment

The electronic variable optical attenuator (EVOA) is a device which controls the output optical power of a system, typically via a feedback loop. EVOAs typically use an opto-mechanical interaction or liquid crysal element to achieve the desired attenuation. EVOAs are available with all combination of fiber and connector type. However, in the current QKD system design, the "decoy state generator" module contains only SMF, and will subsequently be the fiber type used in this EVOA module.

The operational characteristics are as follows:
- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

The only significant modification to the optical message will be the amplitude (power).

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\,\omega_o t}\,e^{i\,\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the EVOA Module

```
Ein1 := Eo1 (* electric field input into port 1 *)
Ein2 := Eo2 (* electric field input into port 2 *)
```

The following parameter values are from or are modeled after COTS EVOAs offered by Oz Optics (http://www.ozoptics.com/ALLNEW_PDF/DTS0010.pdf).

<u>Optical Specifications</u>

```
InsertLoss := 1.0 (* typical mimimum loss in the device, units of -dB *)
AttenuationMin := InsertLoss (* minimum selectable attenuation, units of -dB *)
AttenuationMax := 0.1 dB (* maximum selectable attenuation, units of -dB *)
AttenuationResolution := Tol (* tolerance on the specified attenuation,
increases from (0.1dB @ 1dB) to (1dB @ 25dB), units of +/- dB *)
AttenuationSpeed := 100 * 10^-3 (* quantity of time to change attenuation by 3 dB,
quantity is appx, units of seconds *)
RetLoss :=  -60 (* typical return loss,
signal reflected by an input beam, units of -dB *)
TempH := 70 (* max operational temperature, units of °C *)
TempL := -5 (* min operational temperature, units of °C *)
MaxPwr := 2000 (* maximum operational power, units of mW *)

Repeatto10dB := 0.03
(* repeatability of attenuation setting up to -10dB, units of +/-dB *)
Repeatto30dB := 0.1
(* repeatability of attenuation setting from -10 to -30dB, units of +/-dB *)
Repeatto40dB := 0.3
(* repeatability of attenuation setting from -30 to -40dB, units of +/-dB *)
Repeatto55dB := 0.5
(* repeatability of attenuation setting from -40 to -55dB, units of +/-dB *)
Repeatto60dB := 1.0
 (* repeatability of attenuation setting from -55 to -60dB, units of +/-dB *)
```

Electical/Physical Specifications

```
LatchingType := yes
(* device will remain at selected attenuation when voltage is removed *)
ControlVoltage := 5 (* voltage to active stepper motor *)
PowerConsumption := 150 (* typical electrical power consumption, units of mW *)
```

Attenuation Calculations for EVOA

This EVOA is latching, meaning that it holds the attenuation to which it was last adjusted. Thus, for the module we have to define a "CurrentAttenuation" that will be held until it is again adjusted. We will assume that the rate of adjustment is 30 dB/s for the first generation module. It is stated (on the website) that this rate can change depending on the level of attenuation, and we may wish to include this in the future. There are two ways that we can handle the attenuation adjustment: **1.)** in the electrical message we can supply the duration of the voltage application, or **2.)** work the code to recognize the Δt from the applicaiton of the drive voltage, and continuously adjust the attenuation until the receipt of a new electrical message with "zero" voltage. For this module we will consider -5 V to drive the attenuation to lower values.

Psuedocode for the first option:

```
CurrentAttenuation := 0 (* units of dB *)
AttenuationRate := 30 (* units of dB/second *)
```

Received information from electrical message:

```
ControlVoltage := 5.0 (* units of Volts *)
VoltageDuration := 0.01 (* units of seconds *)
```

Calculating drive direction (psuedocode):

```
if ControlVoltage = 5.0
        DriveDirection = 1.0;
else if ControlVoltage = -5.0
        DriveDirection = -1.0;
else if ControlVoltage = 0.0
        DriveDirection = 0.0
```

```
DriveDirection := 1.0 (* placed for use as an example *)

CurrentAttenuation =
 CurrentAttenuation + DriveDirection * (VoltageDuration * AttenuationRate)
```

$$\texttt{Eout2[Ein1\_, CurrentAttenuation\_] := Ein1} * \sqrt{10^{-\texttt{CurrentAttenuation}/10}}$$

```
(* case: optical input on port 1 *)
Eout1[Ein2_, CurrentAttenuation_] :=
```

$$\texttt{Ein2} * \sqrt{10^{-\texttt{CurrentAttenuation}/10}} \texttt{ (* case: optical input on port 2 *)}$$

Psuedocode for the second option:

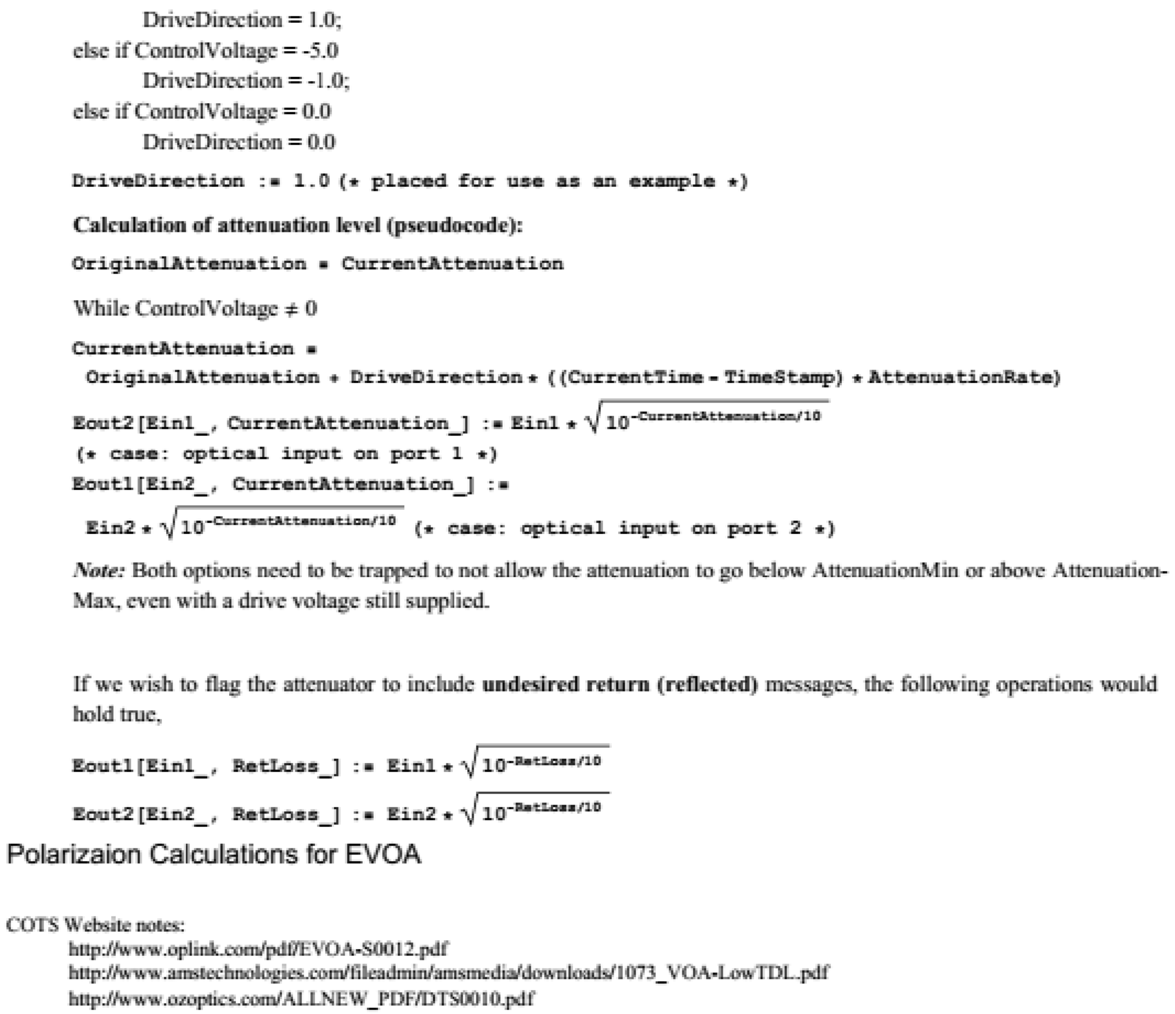
```
CurrentAttenuation := 0 (* units of dB *)
AttenuationRate := 30 (* units of dB/second *)
```

Received information from electrical message:

```
ControlVoltage := 5.0 (* units of Volts *)
TimeStamp := time (* message arrival time *)
```

Calculating drive direction (psuedocode):

```
if ControlVoltage = 5.0
```



```
          DriveDirection = 1.0;
else if ControlVoltage = -5.0
          DriveDirection = -1.0;
else if ControlVoltage = 0.0
          DriveDirection = 0.0
DriveDirection := 1.0 (* placed for use as an example *)
```

Calculation of attenuation level (pseudocode):

```
OriginalAttenuation = CurrentAttenuation

While ControlVoltage ≠ 0

CurrentAttenuation =
 OriginalAttenuation + DriveDirection * ((CurrentTime - TimeStamp) * AttenuationRate)

Eout2[Ein1_, CurrentAttenuation_] := Ein1 * $\sqrt{10^{-CurrentAttenuation/10}}$
(* case: optical input on port 1 *)
Eout1[Ein2_, CurrentAttenuation_] :=
 Ein2 * $\sqrt{10^{-CurrentAttenuation/10}}$ (* case: optical input on port 2 *)
```

Note: Both options need to be trapped to not allow the attenuation to go below AttenuationMin or above AttenuationMax, even with a drive voltage still supplied.

If we wish to flag the attenuator to include **undesired return (reflected)** messages, the following operations would hold true,

```
Eout1[Ein1_, RetLoss_] := Ein1 * $\sqrt{10^{-RetLoss/10}}$

Eout2[Ein2_, RetLoss_] := Ein2 * $\sqrt{10^{-RetLoss/10}}$
```

Polarizaion Calculations for EVOA

COTS Website notes:

http://www.oplink.com/pdf/EVOA-S0012.pdf
http://www.amstechnologies.com/fileadmin/amsmedia/downloads/1073_VOA-LowTDL.pdf
http://www.ozoptics.com/ALLNEW_PDF/DTS0010.pdf

H.9 DEVS MS4ME Derived Pseudocode

```
EVOA - START PASSIVE

EVOA - START PASSIVE EXTERNAL EVENT
  adjust clock based upon time elapsed since last event
  check for existing optical  message
    remove each pulse in bag and store it into a queued optical pulse buffer
  check for existing env message
    check if the received temperature exceeds damaged or degraded threshold
  check for existing ctrl message
    generate the response message for an incoming control message
  set up for first reflected pulse if optical pulse arrived
  go to the Reflect phase else go to the Update Attenuation phase

EVOA - START REFLECT OUTPUT EVENT
```

```
  adjust clock based upon time advance
  output pulse

EVOA - START REFLECT INTERNAL EVENT
  identify any pulses that have not yet been reflected and reflect them
  set up pulse to send in Respond phase
  go to the Respond phase

EVOA - START UPDATE ATTENUATION EXTERNAL EVENT
  adjust clock based upon time elapsed since last event
  check for existing optical  message
    remove each pulse in bag and store it into a queued optical pulse buffer
  check for existing env message
    check if the received temperature exceeds damaged or degraded threshold
  check for existing ctrl message
    generate the response message for an incoming control message
  set up for first reflected pulse if optical pulse arrived
  else hold in the Update Attenuation phase for a change in Attenuation

EVOA - START UPDATE ATTENUATION OUTPUT EVENT
  adjust clock based upon time advance
  output the response message
  change the current attenuation

EVOA - START UPDATE ATTENUATION INTERNAL EVENT
  set up pulse to send in Respond phase if optical pulse arrived
 go to Respond if optical pulse arrived
else go to Update Attenuation if a control message arrived
else go to Passive

 EVOA - START RESPOND EXTERNAL EVENT
  adjust clock based upon time elapsed since last event
  adjust queue
    set a flag that the Respond phase was interrupted by an external event
  check for existing optical  message
    remove each pulse in bag and store it into a queued optical pulse buffer
  check for existing env message
    check if the received temperature exceeds damaged or degraded threshold
  check for existing ctrl message
    generate the response message for an incoming control message
  set up for first reflected pulse if optical pulse arrived
  go to the Reflect phase
    else go to the Update Attenuation phase

EVOA - START RESPOND OUTPUT EVENT
  adjust clock based upon time advance
  adjust pulse amplitude if damaged or degraded
  output pulse to the correct port

EVOA - START RESPOND INTERNAL EVENT
  adjust queue
  check if there any pulses remaining in queue
    set up the next queued pulse
    pulses remaining, so go to Respond
  else check to see if the attenuation is changing
    attenuation changing, go to Update Attenuation
  else go to Passive phase

EVOA – END PASSIVE
```

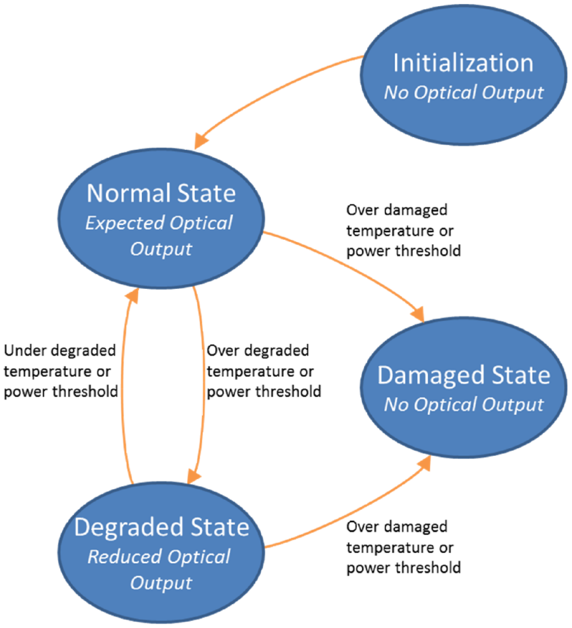
H.10 Component Use Case

H.10.1 Respond to an Optical Packet in the Electronically Controlled Optical Attenuator (EVOA)

Optical packet arrives at EVOA. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the EVOA. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated optical output signal based on the input signal and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
    - Respond to a control message
    - React to an environmental message

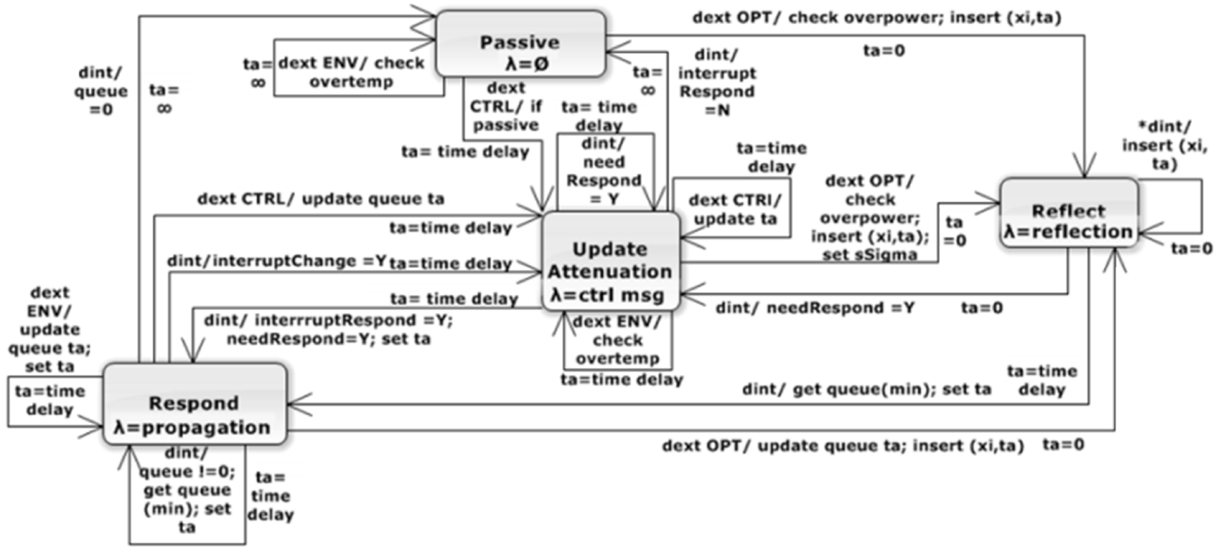
- Assumptions
    - Component has completed initialization sequence at least once
    - Reflections are not affected by component state
    - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 73. Component states.

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, interruptUpdate, needRespond, currentAttenuation, newAttenuation, queue.x1..xn}



* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 74. EVOA phase transition diagram.

H.10.2 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.
- Optical packet attenuated properly to the limit of accuracy.
- Optical packet propagated out the correct port at the correct time.

H.10.3 Respond to an Environmental Packet in the EVOA

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

H.10.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

H.10.5 Respond to a Control Message in the EVOA

Control Message arrives at the component. Component decodes message properly. Records change in condition or state, if applicable. Change component function if in degraded or damaged state or by change in component condition, if necessary.

- Assumptions
    - Component has completed initialization sequence at least once

H.10.6 Respond to Control Message End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

H.11 EVOA Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical

SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 6. *EVOA Test Cases.*

		Inject Ports					Running Totals		
Phase	**Case**	**Opt1**	**Opt2**	**Ctrl**	**Env**	**Notes**	**opt #**	**env #**	**ctrl #**
Passive	1	1	0	0	0	single	1	0	0
	2	0	1	0	0	single	2	0	0
	3	0	0	1	0	single	2	0	1
	4	0	0	0	1	single	2	1	1
	5	1	1	0	0	same time	4	1	1
	6	1	0	1	0	same time	5	1	2
	7	1	1	0	0	differ time	7	1	2
	8	1	0	1	0	differ time	8	1	3
	9	1	1	1	1	same time	10	2	4
	10	1	1	1	1	differ time	12	3	5
	11	0	1	0	1	same time	13	4	5
	12	0	1	0	1	differ time	14	5	5
	13	0	0	1	1	same time	14	6	6
	14	0	0	1	1	differ time	14	7	7
	15	1	0	0	1	same time	15	8	7
	16	1	0	0	1	differ time	16	9	7
	20	2	0	0	0	same time	18	9	7
	21	0	2	0	0	same time	20	9	7
	22	2	1	0	0	same time	23	9	7
	23	2	0	1	0	same time	25	9	8
	24	2	0	0	1	same time	27	10	8
	25	2	0	1	0	differ time	29	10	9
	26	2	1	1	1	same time	32	11	10
	27	2	1	1	1	differ time	35	12	11
	28	0	2	0	1	same time	37	13	11
	29	0	2	0	1	differ time	39	14	11
	30	0	0	1	1	same time	39	15	12
	31	0	0	1	1	differ time	39	16	13
	32	2	0	0	1	same time	41	17	13
	33	2	0	0	1	differ time	43	18	13
totals		27	16	13	18				
Respond	41	2	0	0	0	single	45	18	13
	42	1	1	0	0	single	47	18	13
	43	1	0	1	0	single	48	18	14
	44	1	0	0	1	single	49	19	14
	45	2	1	0	0	same time	52	19	14

	46	2	0	1	0	same time	54	19	15
	47	2	0	0	1	differ time	56	20	15
	48	2	0	1	0	differ time	58	20	16
	49	2	1	1	1	same time	61	21	17
	50	2	1	1	1	differ time	64	22	18
	51	1	1	0	1	same time	66	23	18
	52	1	1	0	1	differ time	68	24	18
	60	3	0	0	0	same time	71	24	18
	61	1	2	0	0	same time	74	24	18
	62	3	1	0	0	same time	78	24	18
	63	3	0	1	0	same time	81	24	19
	64	3	0	0	1	same time	84	25	19
	65	3	0	1	0	differ time	87	25	20
	66	3	1	1	1	same time	91	26	21
	67	3	1	1	1	differ time	95	27	22
	68	1	2	0	1	same time	98	28	22
	69	1	2	0	1	differ time	101	29	22
totals		43	15	9	11				
Math	TC1	1	0	1	2	same time	102	31	23
	TC2	1	0	1	2	same time	103	33	24
	TC3	1	0	1	2	same time	104	35	25
	TC4	1	0	1	2	same time	105	37	26
	TC5	1	0	1	2	same time	106	39	27
	TC6	1	0	1	2	same time	107	41	28
	TC7	1	0	0	2	same time	108	43	28
totals		7	0	6	14				

Notes: 8 - Set attenuation message, set newattenuation = 2

10 - Get attenuation message

13 - Increase attenuation message

14 - Decrease attenuation message

23 - INIT control message sent; OPT1 & Ctrl - same time - Passive: downstream received packets = 214

30 - INIT control message sent - Ctrl & ENV - same time - Passive: downstream received packets = 214

63 - INIT control message sent - OPT1 & Ctrl - same time - Respond: downstream received packets = 211

65 - Set attenuation message, set newattenuation = 5

67 - INIT control message sent - OPT1, OPT2, Ctrl & ENV - differ time - Respond: downstream received packets = 207

H.12 References

OZOptics. (2013). Electrically controlled variable fiber optic attenuator. Retrieved October 2, 2013, from http://www.ozoptics.com/ALLNEW_PDF/DTS0010.pdf

ThorLabs. (2013). Variable fiber optical attenuators, single mode. Retrieved October 2, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6161

Appendix I - Fixed Optical Attenuator (FOA)

I.1 Device Description:

A Fixed Optical Attenuator (FOA) is one of the most basic devices used in fiber optic systems. The FOA is used to attenuate the power of optical signals by a fixed amount, usually expressed in decibels (dBs). FOAs may be fabricated using a number of different principles in order to achieve the desired attenuation. FOAs are typically fabricated using either doped fibers or misaligned splices since both of these are reliable and inexpensive. *Inline*-style FOAs are incorporated into patch cables at the time of manufacture and provide a convenient means of providing a desired fixed attenuation in a fiber link. The alternative *build out*-style attenuator is a small male-female adapter that can be used to adjust the level of attenuation by coupling one or more FOAs between fiber cables. In some cases where variable amounts of optical attenuation are needed, a Variable Optical Attenuator (VOA) may be used which can be manually or electrically adjusted to achieve the desired attenuation level. However, for purposes of this discussion we only consider FOAs. See Figure 1 for an example of a FOA.

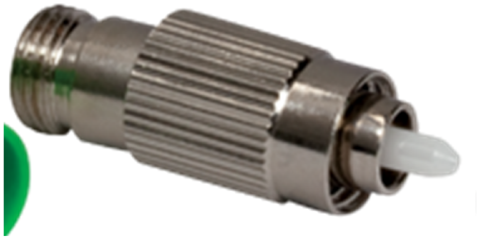


Figure 75. Fixed Optical Attenuator (ThorLabs, 2013).

The FOA is perhaps one of the simplest optical devices to understand and behaviorally model. For this reason, we now present the explicit modeling of a fixed optical attenuator as a means to demonstrate and exercise the Discrete Event Simulation Specification (DEVS) formalism used to represent the behavior of system components. Structurally, the FOA is a bidirectional optical component with two optical ports. Optical signals arriving at one of the ports are attenuated by a fixed amount and then propagated to the other port after a defined propagation delay. The FOA is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the FOA may become permanently damaged which changes its attenuation characteristics. Similarly, the FOA is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the FOA may become temporarily degraded or permanently damaged which changes its attenuation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with modeling the FOA is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the FOA. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the FOA to the researcher.

The next step of the modeling effort was to develop a conceptual model of the FOA using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the FOA. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica

worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

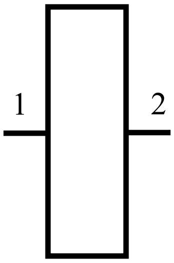


Figure 76. Symbol for Fixed Optical Attenuator (FOA) in the QKD system architecture.

I.2 Fixed Optical Attenuator (FOA) Conceptual Model

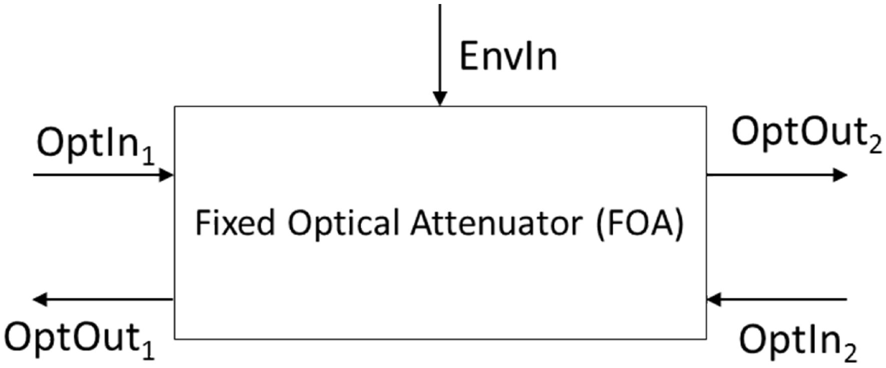


Figure 77. Fixed Optical Attenuator (FOA) conceptual model.

The conceptual model for a FOA consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the FOA. In comparison to the FOA symbol used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the FOA, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The FOA must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the FOA can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the attenuation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

I.3 Mathematical Model

For a detailed mathematical description of the FOA, refer to Section 7.8 which contains the Mathematica worksheet provided by the optical physics SME.

I.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the FOA.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Remove the packet from the queue, calculate the attenuated output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

I.5 Phase Transition Diagram

The phase transition diagram in Fig. 3 shows the phases of the attenuator in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the attenuator at the same time.

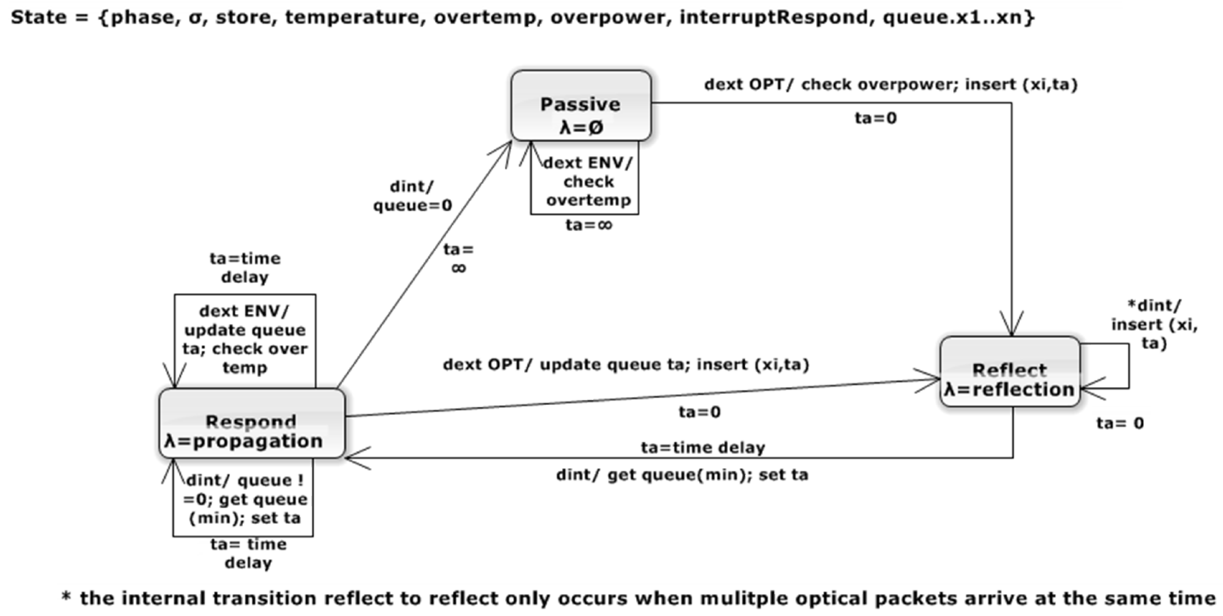

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/
check
overtemp
ta=∞
dint/
queue=0
ta=
∞
ta=time
delay
dext ENV/
update queue
ta; check over
temp
Respond
λ=propagation
dext OPT/ update queue ta; insert (xi,ta)
ta=0
Reflect
λ=reflection
*dint/
insert (xi,
ta)
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue !
=0; get queue
(min); set ta
ta= time
delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 78. FOA phase transition diagram.

I.6 Event-Trace Diagram

This section shows various examples of packets entering the FOA. The tables list the states the attenuator proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the attenuator, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

I.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 32. *Case I state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (x*i*, *tp*)**	**Notes: assume tp=5**
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	
5	s3	entry	passive	inf	x1	c	n	n	n	null	

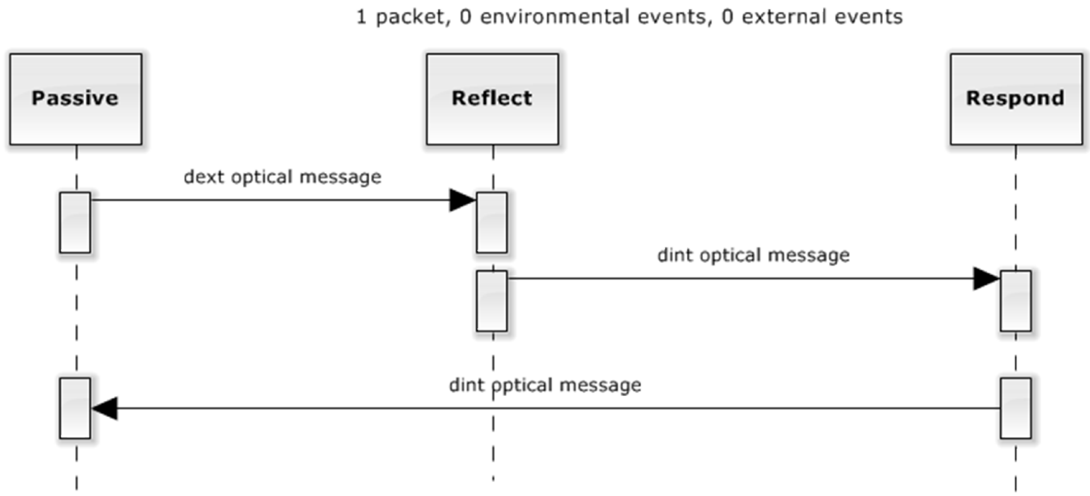


Figure 79. Case I sequence diagram.

I.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 33. *Case II state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	Interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

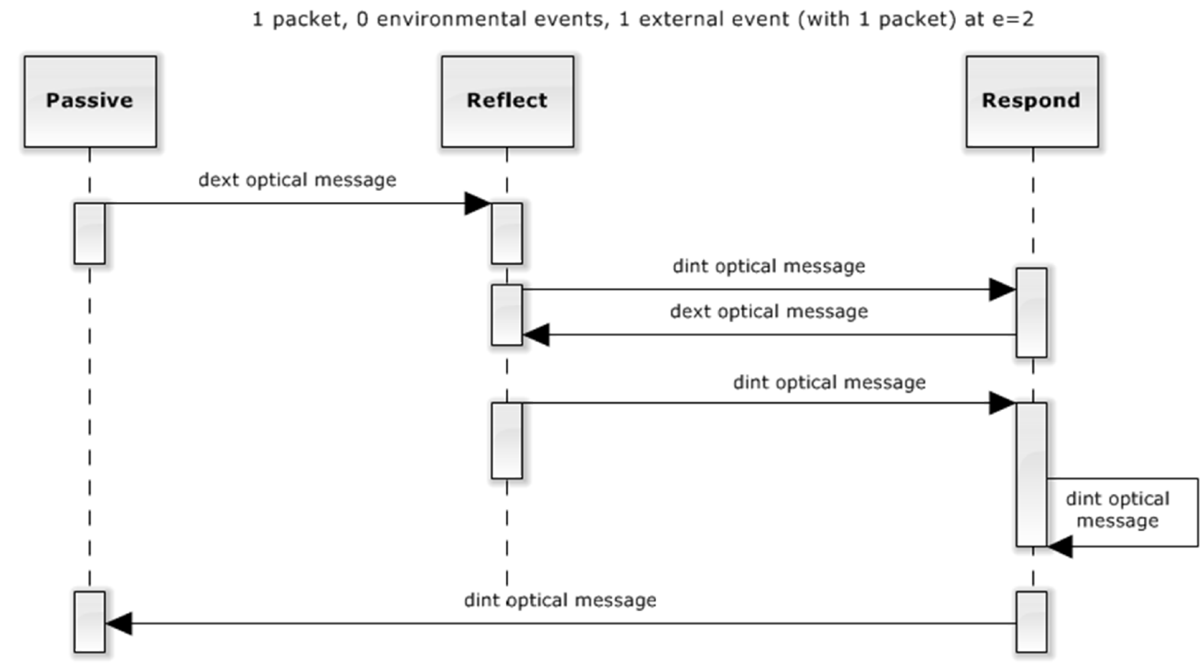


Figure 80. Case II sequence diagram.

I.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 34. *Case III state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	

3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

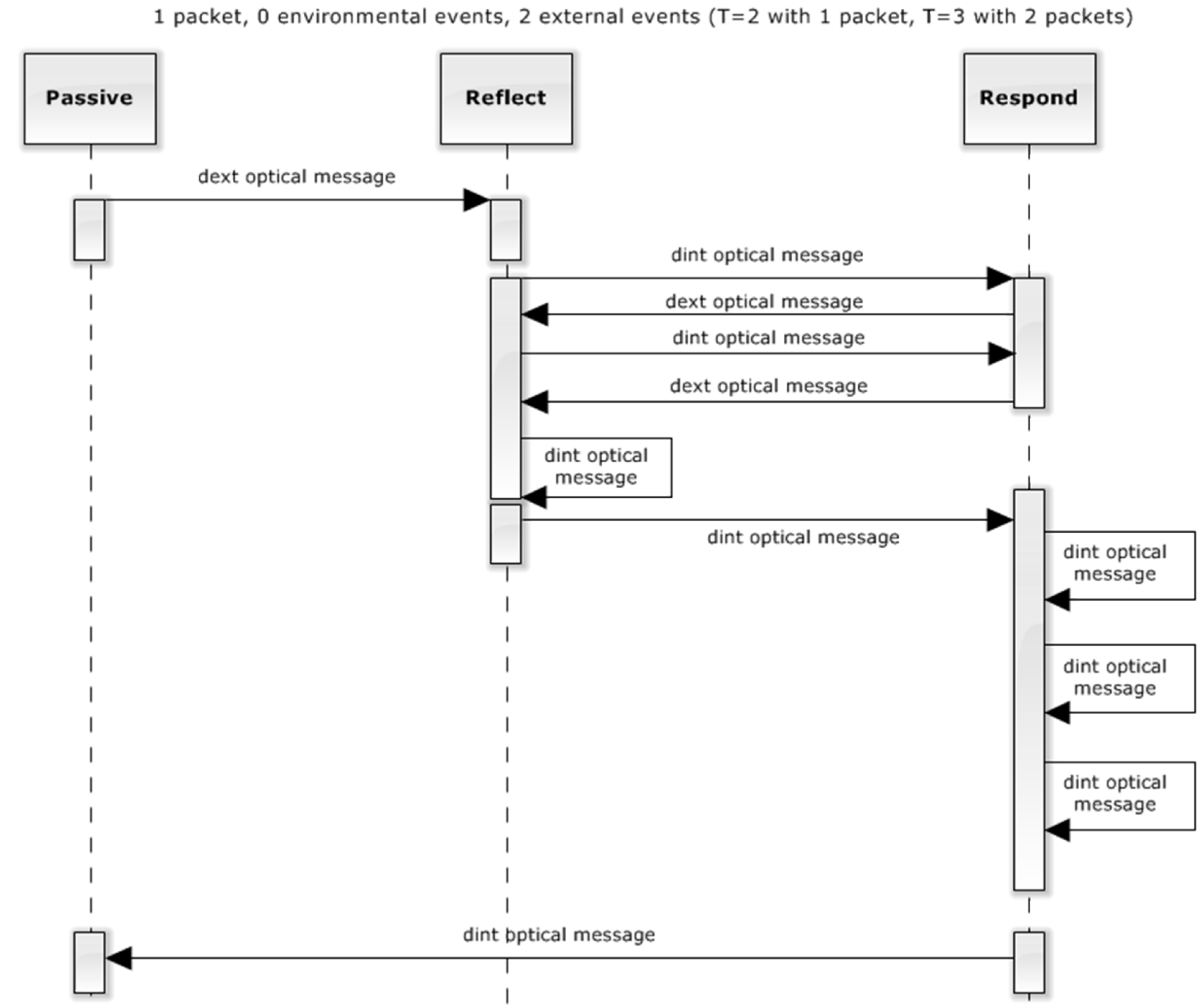


Figure 81. Case III sequence diagram.

I.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 35. *Case IV state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

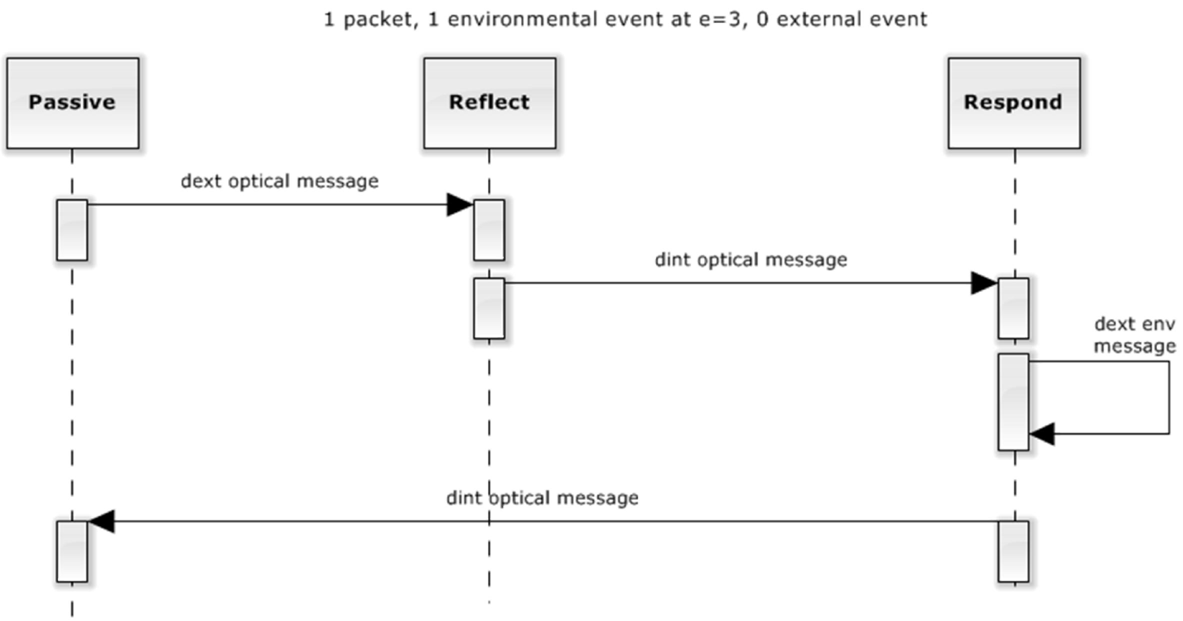


Figure 82. Case IV sequence diagram.

I.7 Fixed Optical Attenuator (FOA) Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.

- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, "store", temperature, "overtemp", "overpower", "interruptRespond", queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level
"overpower" = flag variable set when device meets or exceeds damaged optical power level
"interruptRespond" = flag variable set when device is interrupted by an external event
Peak power = full width, half maximum power calculation of the pulse

For the fixed attenuator we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{attenuator}$* **= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {"passive", "reflect", "respond"}
overtemp = flag variable set when device meets or exceeds damaged temperature level

overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time delay$_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay$_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"OptIn$_1$", "OptIn$_2$", "EnvIn"} with
X_M = {("OptIn$_1$", V_{opt}), ("OptIn$_2$", V_{opt}), ("EnvIn", V_{env})} is the set of input ports and values.

OutPorts = {"OptOut$_1$", "OptOut$_2$"} with
Y_M = {("OptOut$_1$", $Y_{OptOut1}$), ("OptOut$_2$", $Y_{OptOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond, queue*} = {{"passive", "reflect", "respond"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x {"Y","N"} x *V*}

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue, e, (($p_i$,$v_i$),….
                                                                                        ($p_n$,$v_n$))) =
("reflect", 0, store, temperature, overtemp, overpower,interruptRespond,  queue.x1..xn)
  if phase = "passive" and $p \in$ {"OptIn$_1$", "OptIn$_2$"}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower =  "Y"
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = "N"

("reflect", 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = "respond" and $p \in$ {"OptIn$_1$", "OptIn$_2$"}
    update_delay(queue)
    for messagebag != null
     current = messagebag_first()
     if current.value.power > damaged.power
       overpower =  "Y"
     insert_event_q(current)
      remove_event_m(current)
   queue.current = queue_need_reflected()
```

```
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
   interruptRespond= “Y”
   timeLeftRespond = timeLeftRespond - e

 (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                        queue.x1..xn)
   if phase = “respond” and p = “EnvIn”
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”
     time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
(“reflect”, 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
  if phase = “reflect” and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)

 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
  if phase = “reflect” and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = “N”
      current = queue_min()
      time.delay = current.time.delay
      if InPort = “OptIn$_1$”
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_2$”
      if InPort = “OptIn$_2$”
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_1$”
      timeLeftRespond = propagation delay
    else
      time.delay = timeLeftRespond

  (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if InPort = “OptIn$_1$”
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_2$”
      if InPort = “OptIn$_2$”
        outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_1$”
      interruptRespond= “N”

  (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size  = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

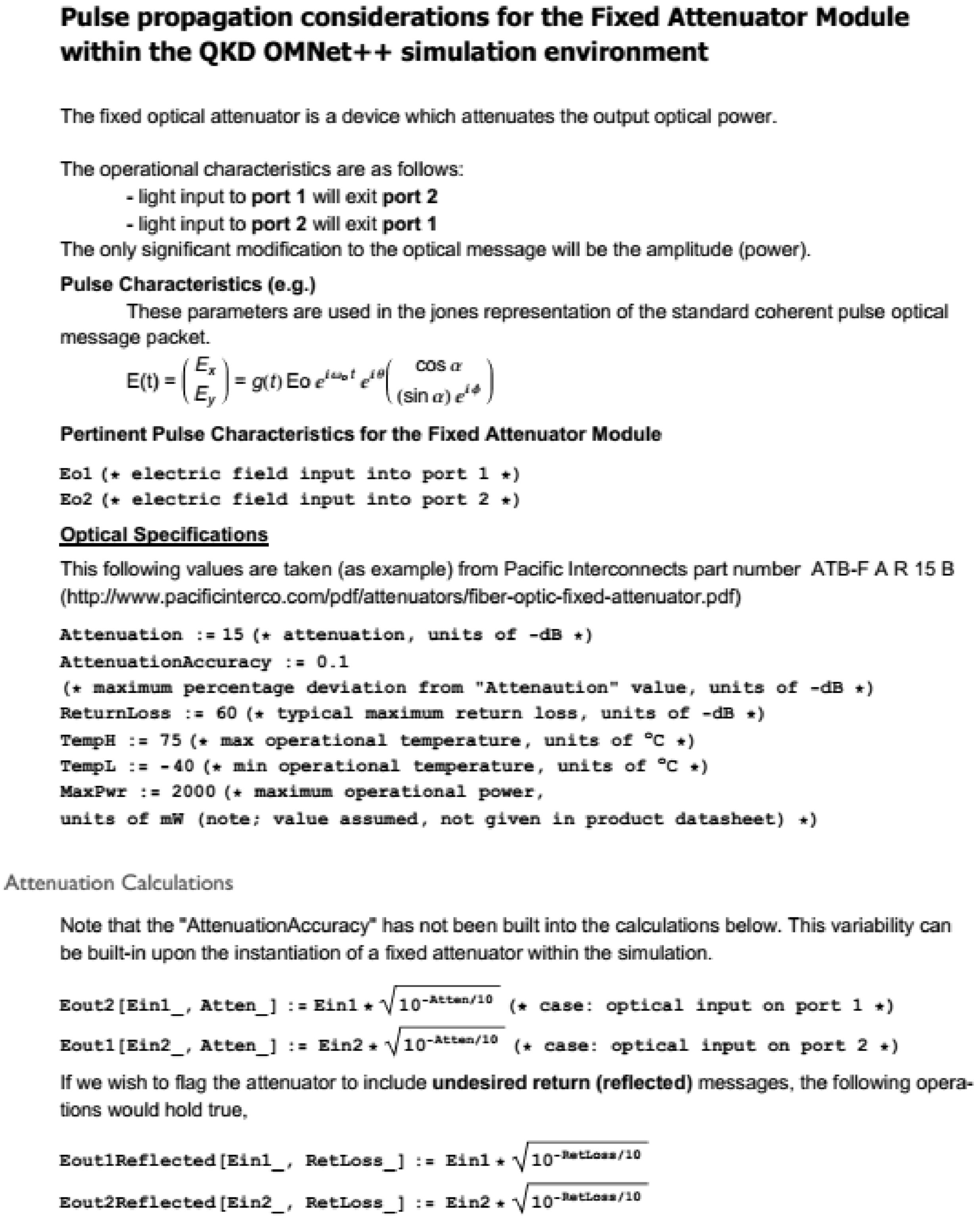
Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (output.port, output.pulse)
    if phase = “respond”

  $\emptyset$ (null output)
   otherwise;
```

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower, interruptRespond, queue) = \sigma$;

I.8 Mathematical model



Pulse propagation considerations for the Fixed Attenuator Module within the QKD OMNet++ simulation environment

The fixed optical attenuator is a device which attenuates the output optical power.

The operational characteristics are as follows:
- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

The only significant modification to the optical message will be the amplitude (power).

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\omega_o t}\,e^{i\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Fixed Attenuator Module

```
Eo1 (* electric field input into port 1 *)
Eo2 (* electric field input into port 2 *)
```

<u>Optical Specifications</u>

This following values are taken (as example) from Pacific Interconnects part number ATB-F A R 15 B (http://www.pacificinterco.com/pdf/attenuators/fiber-optic-fixed-attenuator.pdf)

```
Attenuation := 15 (* attenuation, units of -dB *)
AttenuationAccuracy := 0.1
(* maximum percentage deviation from "Attenaution" value, units of -dB *)
ReturnLoss := 60 (* typical maximum return loss, units of -dB *)
TempH := 75 (* max operational temperature, units of °C *)
TempL := -40 (* min operational temperature, units of °C *)
MaxPwr := 2000 (* maximum operational power,
units of mW (note; value assumed, not given in product datasheet) *)
```

Attenuation Calculations

Note that the "AttenuationAccuracy" has not been built into the calculations below. This variability can be built-in upon the instantiation of a fixed attenuator within the simulation.

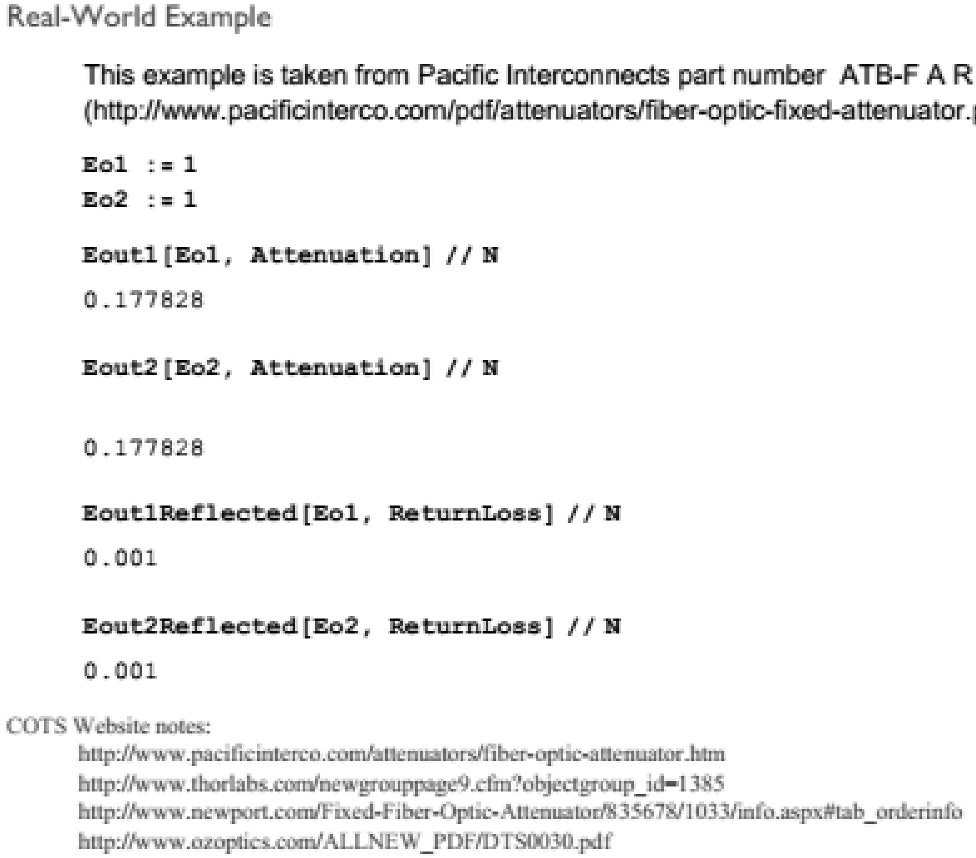
Eout2[Ein1_, Atten_] := Ein1 * $\sqrt{10^{-\mathrm{Atten}/10}}$ (* case: optical input on port 1 *)

Eout1[Ein2_, Atten_] := Ein2 * $\sqrt{10^{-\mathrm{Atten}/10}}$ (* case: optical input on port 2 *)

If we wish to flag the attenuator to include **undesired return (reflected)** messages, the following operations would hold true,

Eout1Reflected[Ein1_, RetLoss_] := Ein1 * $\sqrt{10^{-\mathrm{RetLoss}/10}}$

Eout2Reflected[Ein2_, RetLoss_] := Ein2 * $\sqrt{10^{-\mathrm{RetLoss}/10}}$



Real-World Example

This example is taken from Pacific Interconnects part number ATB-F A R 15 B (http://www.pacificinterco.com/pdf/attenuators/fiber-optic-fixed-attenuator.pdf) as given above.

```
Eo1 := 1
Eo2 := 1

Eout1[Eo1, Attenuation] // N

0.177828

Eout2[Eo2, Attenuation] // N


0.177828

Eout1Reflected[Eo1, ReturnLoss] // N

0.001

Eout2Reflected[Eo2, ReturnLoss] // N

0.001
```

COTS Website notes:

http://www.pacificinterco.com/attenuators/fiber-optic-attenuator.htm
http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1385
http://www.newport.com/Fixed-Fiber-Optic-Attenuator/835678/1033/info.aspx#tab_orderinfo
http://www.ozoptics.com/ALLNEW_PDF/DTS0030.pdf

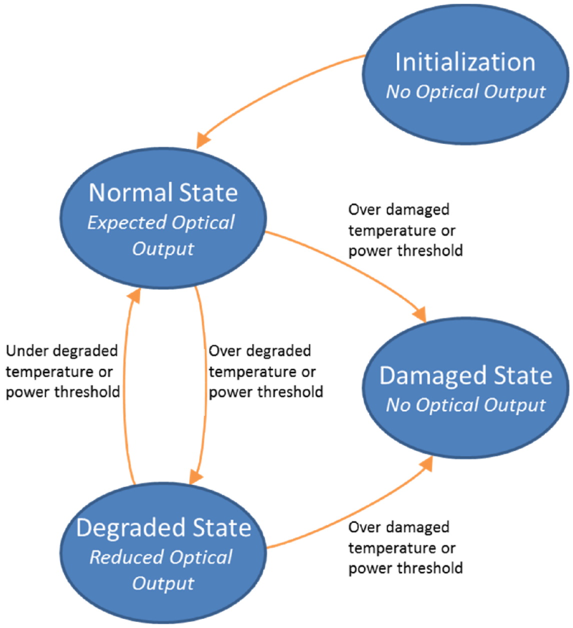
I.9 Component Use Case

I.9.1 Respond to an Optical Packet in the Fixed Optical Attenuator (FOA)

Optical packet arrives at the FOA. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the FOA. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated optical output signal based on the input signal and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
    - React to an environmental message

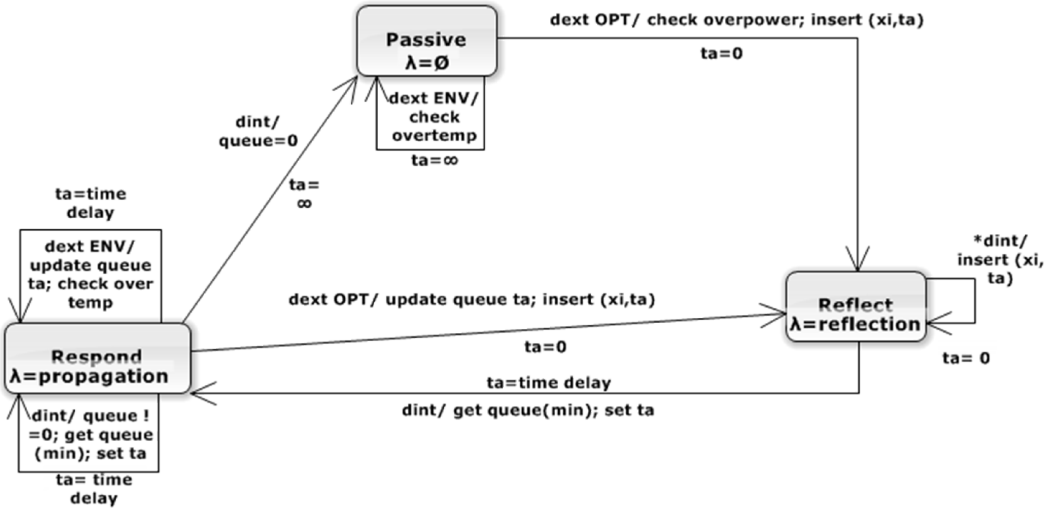
- Assumptions
    - Component has completed initialization sequence at least once
    - Reflections are not affected by component state
    - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 83. Component states.

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}


Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
Respond
λ=propagation
ta=0
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue ! =0; get queue (min); set ta
ta= time delay

* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 84. FOA phase transition diagram.

I.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.
- Optical packet attenuated properly to the limit of accuracy.
- Optical packet propagated out the correct port at the correct time.

I.9.3 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.
- Optical packet attenuated properly to the limit of accuracy.
- Optical packet propagated out the correct port at the correct time.

I.9.4 Respond to an Environmental Packet in the Fixed Optical Attenuator (FOA)

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
  - None

I.9.5 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

I.10 Fixed Attenuator Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Fixed Attenuator Test Cases.*

		Inject Ports				Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Env**	**Notes**	**opt #**	**env #**
Passive	1	1	0	0	single	1	0
	2	0	1	0	single	2	0
	3	0	0	1	single	2	1
	4	1	1	0	same time	4	1

	5	1	1	0	differ time	6	1
	6	1	1	1	same time	8	2
	7	1	1	1	differ time	10	3
	8	0	1	1	same time	11	4
	9	0	1	1	differ time	12	5
	10	1	0	1	same time	13	6
	11	1	0	1	differ time	14	7
	20	2	0	0	same time	16	7
	21	0	2	0	same time	18	7
	22	2	2	0	same time	22	7
	23	2	2	0	differ time	26	7
	24	2	2	1	same time	30	8
	25	2	2	1	differ time	34	9
	26	0	2	1	same time	36	10
	27	0	2	1	differ time	38	11
	28	2	0	1	same time	40	12
	29	2	0	1	differ time	42	13
totals		21	21	13	42		
Respond	41	2	0	0	single	44	13
	42	0	2	0	single	46	13
	43	1	0	1	single	47	14
	44	2	1	0	same time	50	14
	45	2	1	0	differ time	53	14
	46	2	1	1	same time	56	15
	47	2	1	1	differ time	59	16
	48	0	2	1	same time	61	17
	49	0	2	1	differ time	63	18
	50	2	0	1	same time	65	19
	51	2	0	1	differ time	67	20
	60	3	0	0	same time	70	20
	61	0	3	0	same time	73	20
	62	3	2	0	same time	78	20
	63	3	2	0	differ time	83	20
	64	3	2	1	same time	88	21
	65	3	2	1	differ time	93	22
	66	0	3	1	same time	96	23
	67	0	3	1	differ time	99	24
	68	3	0	1	same time	102	25
	69	3	0	1	differ time	105	26
totals		36	27	13	63		
	TC1	1	0	2	single	106	28
	TC2	1	0	2	single	107	30
	TC3	1	0	2	single	108	32

	TC4	1	0	2	single	109	34
	TC5	1	0	2	single	110	36
	TC6	1	0	2	single	111	38
	TC7	1	0	2	single	112	40
totals		7	0	14	21		

I.11 References

ThorLabs. (2013). Fixed fiber optical attenuators. Retrieved October 28, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1385

Appendix J - Half-Wave Plate

J.1 Device Description:

The half-wave plate is an optical device that transforms the input wave by retarding one of the components of the wave. The "slow" axis is retarded by some amount while the "fast" axis has a very small reduction in its transmission velocity through the device. A half wave plate can be used to rotate the polarization of linearly polarized light with very little loss (Saleh & Teich, 1991). The change of angle is twice the difference of input angle and the plate angle of the slow axis but note that if the incoming linearly polarized light falls upon either principal axis, there is no change. See Figure 1 for an example of a half-wave plate.

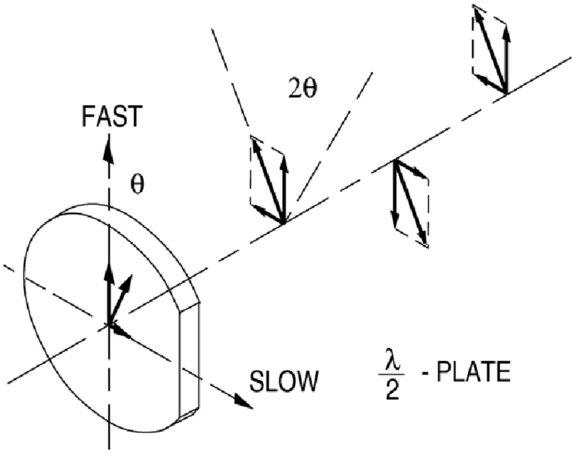


Figure 85. View of a zero-order half-wave plate (Newport, 2013).

A typical half-wave plate is made from two quarter-wave plates glued together with a free-space gap between them and mounted into some type of stabilizing housing. Although each quarter-wave plate changes linearly polarized light into left or right-circular light, by aligning the fast axis of one quarter-wave plate with the slow-axis of the second pate, the resultant is light that is still linearly polarized, but with a new polarization angle. The air gap between the quarter-wave plates allows for higher levels of optical power through the plates.

The Half-wave plate is a bidirectional optical component with two optical ports. Optical signals arriving at the input port are propagated to the other port after a defined propagation delay and the polarizing material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the Half-wave plate may become permanently damaged which changes its propagation characteristics. Similarly, the Half-wave plate is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the Half-wave plate may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a "normal" operating temperature.

The first step involved with the modeling the Half-wave plate is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the Half-wave plate. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the Half-wave plate to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the Half-wave plate using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the Half-wave plate. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

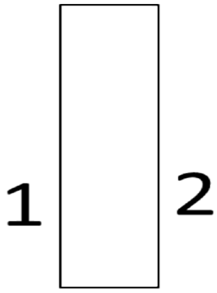


Figure 86. Symbol for the half-wave plate in the QKD system architecture.

J.2 Half-wave plate Conceptual Model

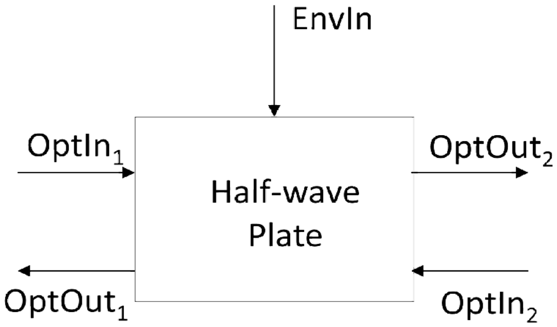


Figure 87. Half-wave plate conceptual model.

The conceptual model for a Half-wave plate consists of two optical input ports {OptIn_1, OptIn_2}, two optical output ports {OptOut_1, OptOut_2}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the Half-wave plate. In comparison to the Half-wave plate symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the Half-wave plate, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 3 will instantaneously generate a reflected emitting out of OptOut_1.

The Half-wave plate calculates changes to the power, the amplitude, ellipticity, polarization and phase of any packet coming through either optical port after a time equaling the propagation delay of the module. The packet is calculated at full power minus some small amount to account for attenuation through the device and retards the packet along the “slow” axis of the plate to some accuracy, dependent on the design of the plate.

The Half-wave plate must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the Half-wave plate can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

J.3 Mathematical Model

For a detailed mathematical description of the Half-wave plate, refer to Section 8.8 which contains the Mathematica worksheet provided by the optical physics SME.

J.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the Half-wave plate.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue
- Immediately calculate the reflected power of the signal and send its output with the same port number.

- Retrieve the input optical signal from the queue, and determine the input port, ellipticity, polarization, and overall phase of the signal.
- Update the values of the input optical signal based on the characteristics of the half-wave plate, the original values of the input optical signal and the current environment.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

J.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the Half-wave plate in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the Half-wave plate at the same time.

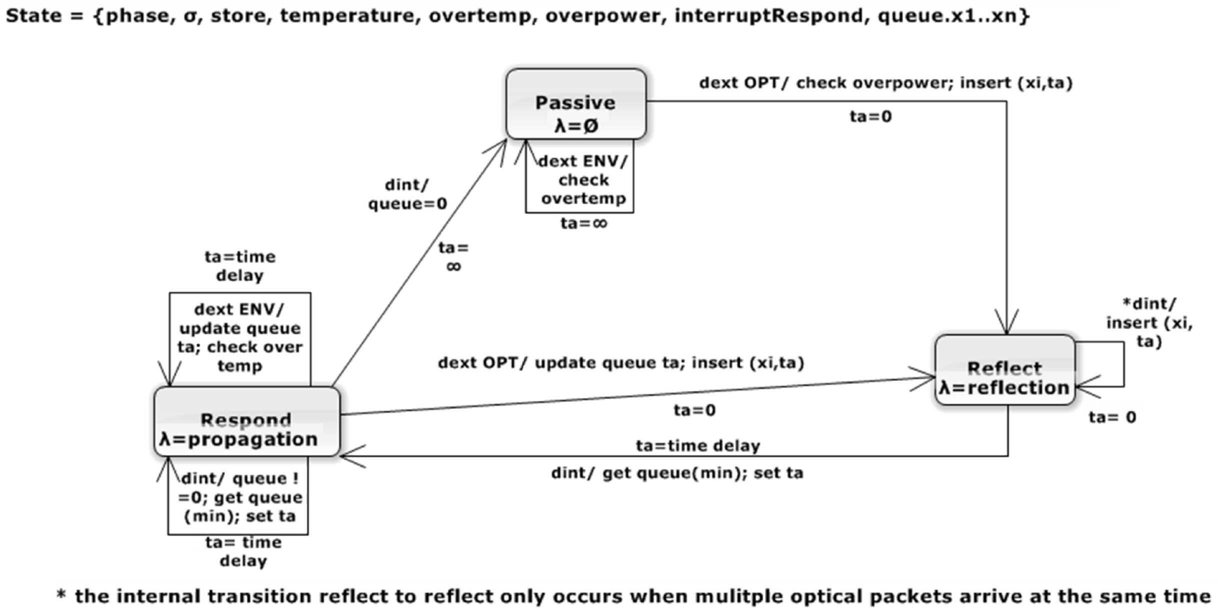

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
dext OPT/ update queue ta; insert (xi,ta)
ta=0
Reflect
λ=reflection
*dint/ insert (xi, ta)
ta= 0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 88. Half-wave plate phase transition diagram.

J.6 Event-Trace Diagram

This section shows various examples of packets entering the FOA. The tables list the states the attenuator proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the attenuator, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable

- Queue: contents of the queue for that state
- Notes: any notes for that state

J.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 36. *Case I state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	
5	s3	entry	passive	inf	x1	c	n	n	n	null	

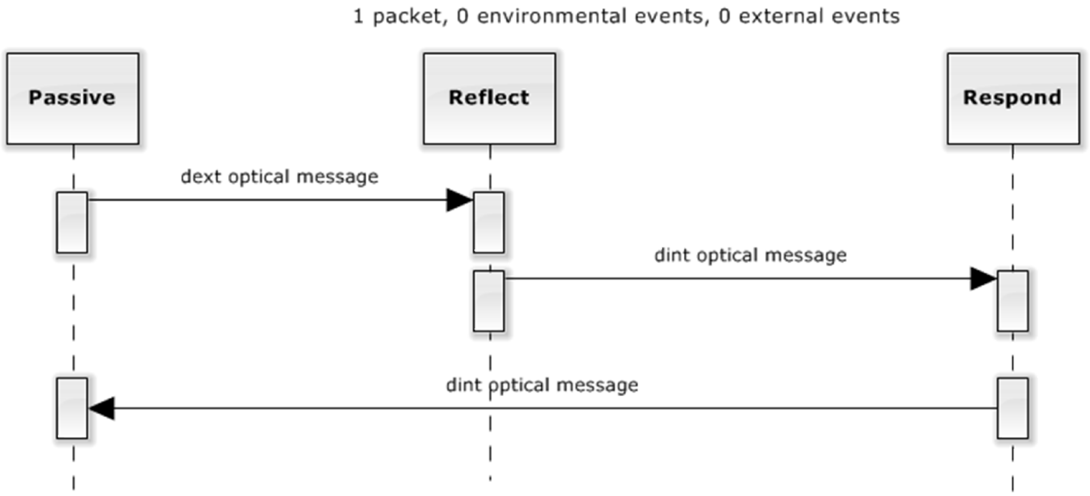


Figure 89. Case I sequence diagram.

J.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 37. *Case II state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	Interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	

0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

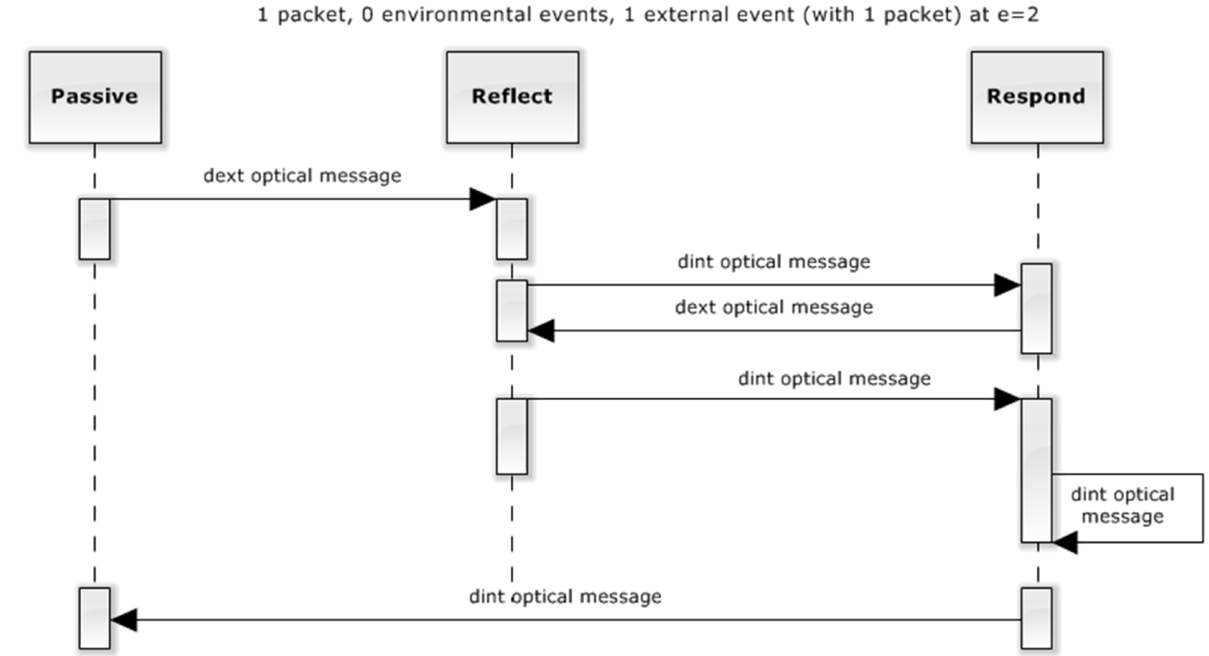


Figure 90. Case II sequence diagram.

J.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 38. *Case III state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	2 opt	0 ctrl							

0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

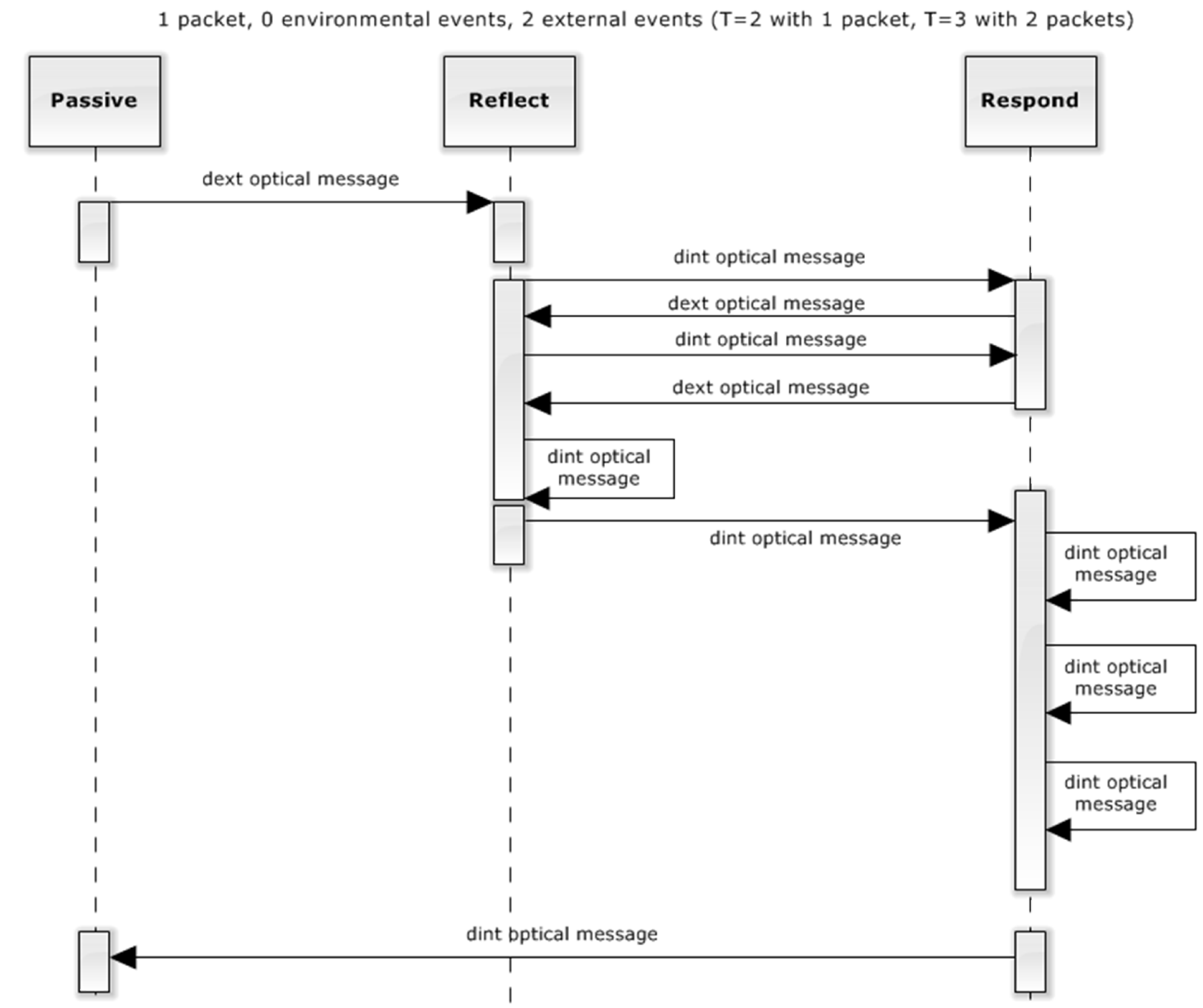


Figure 91. Case III sequence diagram.

J.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 39. *Case IV state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n	n	null	
5	s4	entry	passive	inf	x1	c2	y	n	n	null	

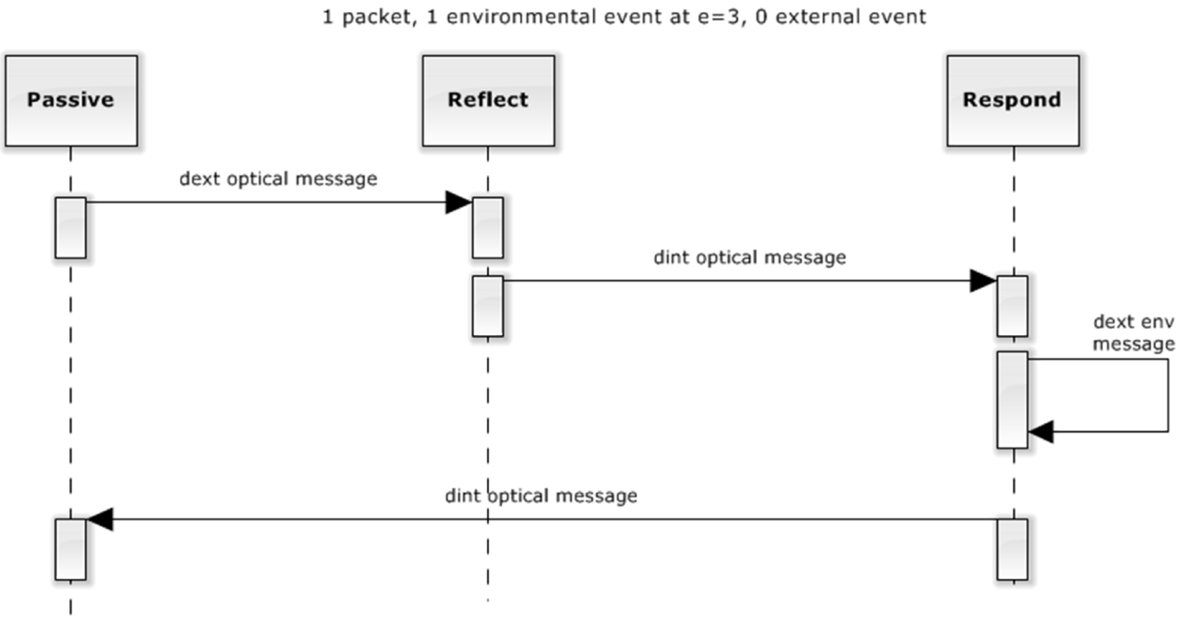


Figure 92. Case IV sequence diagram.

J.7 Half-wave plate Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.

- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
Peak power = full width, half maximum power calculation of the pulse

For the Half-wave plate we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \text{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{\textbf{\textit{Half-wave plate}}}$* **= (*$X_M$, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta*)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “reflect”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level

overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time $delay_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
$q.v_{min}$ = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its x (p,v) values into the variable *store*

InPorts = {“OptIn$_1$”, “OptIn$_2$”, “EnvIn”} with
X_M = {(“OptIn$_1$”, V_{opt}), (“OptIn$_2$”, V_{opt}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“OptOut$_1$”, “OptOut$_2$”} with
Y_M = {(“OptOut$_1$”, $Y_{OptOut1}$), (“OptOut$_2$”, $Y_{OptOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond, queue*} = {{“passive”, “reflect”, “respond”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x *V*}

External Transition Function:

```
δ_ext(phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue, e, ((p_i,v_i),….
                                                                                    (p_n,v_n))) =
(“reflect”, 0, store, temperature, overtemp, overpower,interruptRespond, queue.x1..xn)
  if phase = “passive” and p ∈ {“OptIn1”, “OptIn2”}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower = “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = “respond” and p ∈ {“OptIn1”, “OptIn2”}
    update_delay(queue)
    for messagebag != null
     current = messagebag_first()
     if current.value.power > damaged.power
       overpower = “Y”
     insert_event_q(current)
      remove_event_m(current)
    queue.current = queue_need_reflected()
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
   interruptRespond= "Y"
   timeLeftRespond = timeLeftRespond - e

("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = "passive" and p =$\in$ "EnvIn"
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = "Y"

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                      queue.x1..xn)
   if phase = "respond" and p = "EnvIn"
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"
     time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
("reflect", 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
  if phase = "reflect" and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
  if phase = "reflect" and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = "N"
      current = queue_min()
      time.delay = current.time.delay
      if InPort = "$OptIn_1$"
        outputPulse = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "$OptOut_2$"
      if InPort = "$OptIn_2$"
```

```
        outputPulse = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_1$”
      timeLeftRespond = propagation delay
    else
      time.delay = timeLeftRespond

  (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if InPort = “OptIn$_1$”
        outputPulse = calcPolar (current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_2$”
      if InPort = “OptIn$_2$”
        outputPulse = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “OptOut$_1$”
      interruptRespond= “N”

  (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

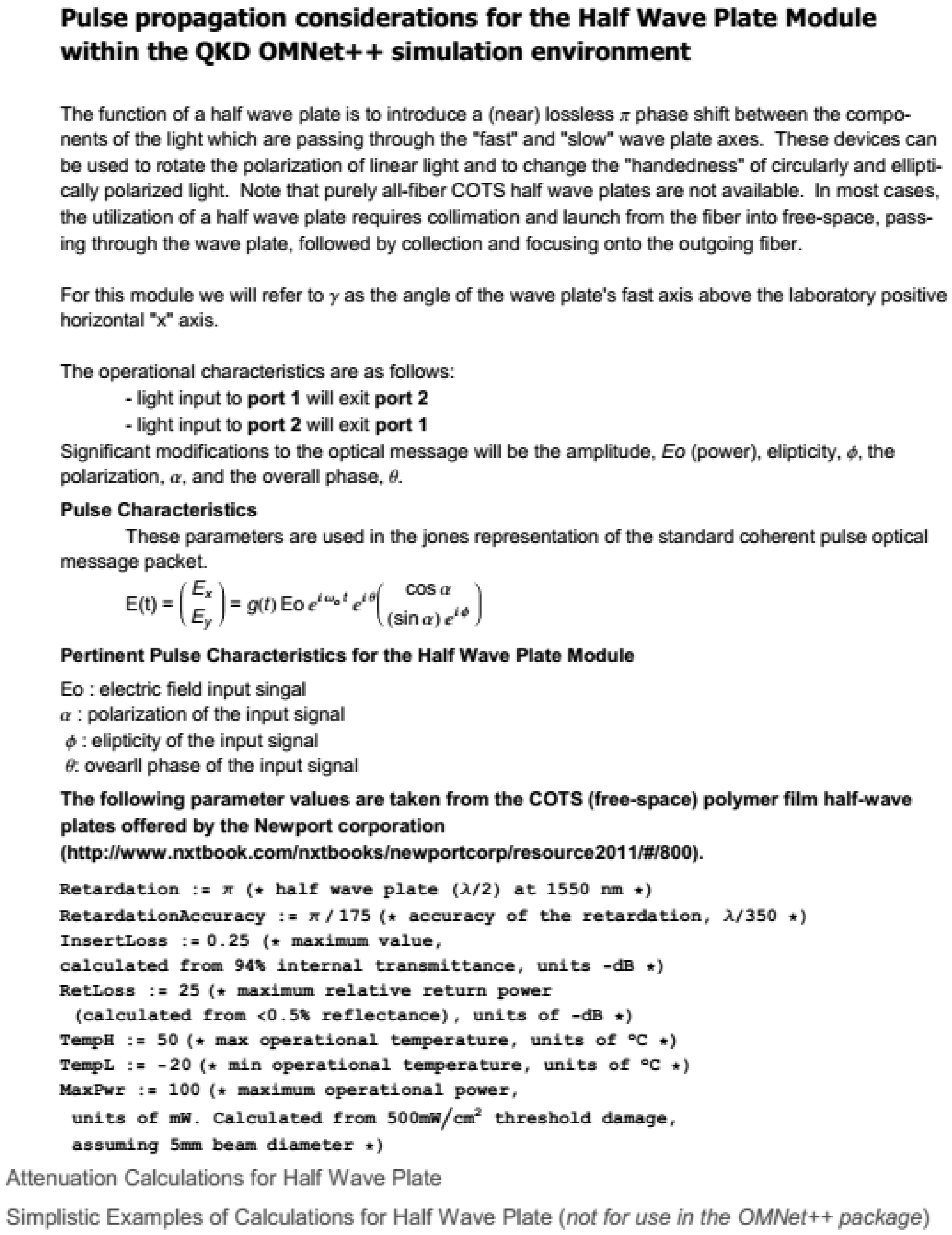
Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (output.port, output.pulse)
    if phase = “respond”

  ∅ (null output)
    otherwise;
```

Time advance Function:

ta(*phase*, σ, *store, temperature*, *overtemp*, *overpower, interruptRespond, queue*) = σ;

J.8 Mathematical Model



Pulse propagation considerations for the Half Wave Plate Module within the QKD OMNet++ simulation environment

The function of a half wave plate is to introduce a (near) lossless π phase shift between the components of the light which are passing through the "fast" and "slow" wave plate axes. These devices can be used to rotate the polarization of linear light and to change the "handedness" of circularly and elliptically polarized light. Note that purely all-fiber COTS half wave plates are not available. In most cases, the utilization of a half wave plate requires collimation and launch from the fiber into free-space, passing through the wave plate, followed by collection and focusing onto the outgoing fiber.

For this module we will refer to γ as the angle of the wave plate's fast axis above the laboratory positive horizontal "x" axis.

The operational characteristics are as follows:

- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

Significant modifications to the optical message will be the amplitude, *Eo* (power), elipticity, ϕ, the polarization, α, and the overall phase, θ.

Pulse Characteristics

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$E(t) = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\omega_o t}\,e^{i\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Half Wave Plate Module

Eo : electric field input singal

α : polarization of the input signal

ϕ : elipticity of the input signal

θ: ovearll phase of the input signal

The following parameter values are taken from the COTS (free-space) polymer film half-wave plates offered by the Newport corporation (http://www.nxtbook.com/nxtbooks/newportcorp/resource2011/#/800).

```
Retardation := π (* half wave plate (λ/2) at 1550 nm *)
RetardationAccuracy := π / 175 (* accuracy of the retardation, λ/350 *)
InsertLoss := 0.25 (* maximum value,
calculated from 94% internal transmittance, units -dB *)
RetLoss := 25 (* maximum relative return power
  (calculated from <0.5% reflectance), units of -dB *)
TempH := 50 (* max operational temperature, units of °C *)
TempL := -20 (* min operational temperature, units of °C *)
MaxPwr := 100 (* maximum operational power,
 units of mW. Calculated from 500mW/cm² threshold damage,
 assuming 5mm beam diameter *)
```

Attenuation Calculations for Half Wave Plate

Simplistic Examples of Calculations for Half Wave Plate (*not for use in the OMNet++ package*)

Proper Mathematical Calculations for use with OMNet++ Package

Please implement the following calculations for polarization, ellipticity, and overall phase in the OMNet++ simulation environment (don't use the calculations in the preceding section)

Note: Refresh system kernel before using the following calculations. *This section is necessary and required for using the example in the final (Full Example) section.*

The output form of **Jthrough** (previous section) needs to be expressed in a manner which can be passed in our standard optical message, namely θ_{out}, α_{out}, and ϕ_{out}. I thank Ramesh for valuable help with these calculations. The general form of the output vector is,

$$\begin{pmatrix} \mathrm{P}\, e^{i\lambda} \\ \mathrm{Q}\, e^{i\delta} \end{pmatrix} = e^{i\,\theta\mathrm{out}} \begin{pmatrix} \mathrm{P} \\ \mathrm{Q}\, e^{i(\delta-\lambda)} \end{pmatrix}$$

where,

$$\mathrm{P}[\gamma\_,\ \alpha\_,\ \phi\_] := \frac{1}{2}\sqrt{2 + 2\,\mathrm{Cos}[2\,\alpha]\ \mathrm{Cos}[4\,\gamma] + 2\,\mathrm{Cos}[\phi]\ \mathrm{Sin}[2\,\alpha]\ \mathrm{Sin}[4\,\gamma]}$$

$$\mathrm{Q}[\gamma\_,\ \alpha\_,\ \phi\_] := \frac{1}{\sqrt{2}}\sqrt{1 - \mathrm{Cos}[2\,\alpha]\ \mathrm{Cos}[4\,\gamma] - \mathrm{Cos}[\phi]\ \mathrm{Sin}[2\,\alpha]\ \mathrm{Sin}[4\,\gamma]}$$

Thus, the output polarization, α_{out}, is simply

$$\alpha\mathrm{out1}[\gamma\_,\ \alpha\_,\ \phi\_] := \mathrm{ArcCos}[\mathrm{P}[\gamma,\ \alpha,\ \phi]]$$

or, equally,

$$\alpha\mathrm{out2}[\gamma\_,\ \alpha\_,\ \phi\_] := \mathrm{ArcSin}[\mathrm{Q}[\gamma,\ \alpha,\ \phi]]$$

To conform to using only one value for the ellipticity in the polarization state let, $\gamma_{\mathrm{out}} = \delta - \lambda$, where

$$\lambda[\gamma\_,\ \alpha\_,\ \phi\_] := \mathrm{ArcTan}\left[\frac{2\,\mathrm{Cos}\left[\frac{\phi}{2}\right]\ \mathrm{Sin}\left[\frac{\phi}{2}\right]\ \mathrm{Sin}[\alpha]\ \mathrm{Sin}[2\,\gamma]}{\mathrm{Cos}[2\,\gamma - \alpha] - 2\left(\mathrm{Sin}\left[\frac{\phi}{2}\right]\right)^2 \mathrm{Sin}[\alpha]\ \mathrm{Sin}[2\,\gamma]}\right]$$

$$\delta[\gamma\_,\ \alpha\_,\ \phi\_] := \mathrm{ArcTan}\left[\frac{-2\,\mathrm{Cos}\left[\frac{\phi}{2}\right]\ \mathrm{Sin}\left[\frac{\phi}{2}\right]\ \mathrm{Sin}[\alpha]\ \mathrm{Cos}[2\,\gamma]}{\left(\mathrm{Sin}[2\,\gamma - \alpha] + 2\left(\mathrm{Sin}\left[\frac{\phi}{2}\right]\right)^2 \mathrm{Sin}[\alpha]\ \mathrm{Cos}[2\,\gamma]\right)}\right]$$ (* Note; this is where the complex zero occurs. The numerator needs to be trapped for zero to make the entire argument of the ArcTan be zero. *)

$$\gamma\mathrm{out}[\gamma\_,\ \alpha\_,\ \phi\_] := \delta[\gamma,\ \alpha,\ \phi] - \lambda[\gamma,\ \alpha,\ \phi]$$

$$\theta\mathrm{out}[\theta\_,\ \gamma\_,\ \alpha\_,\ \phi\_] := \theta + \lambda[\gamma,\ \alpha,\ \phi]$$

Full Example of an Optical Pulse Passed Through the Half Wave Plate

Input optical parameters

```
Ein := Eo (* input light amplitude, arbitrary *)
αin := 1.25 (* input light polarization,
corresponds to an angle of 71.6197 above horizontal *)
ϕin := π / 24 (* input light ellipticity,
corresponds to slightly elliptical light *)
θin := 0 (* input light overall phase *)
```

Half wave plate optical parameters

```
InsertLoss := 0.25 (* intrinsic optical loss, units of -dB *)
γplate := π / 4 (* half wave plate fast axis orientation,
  corresponds to 45° above horizontal *)
```

Output electric field amplitude:

$$\text{Eout[Ein\_] = Ein} * \sqrt{10^{-\text{InsertLoss}/10}} \text{ // N}$$

```
0.944061 Eo
```

Output polarization (α**out1** and α**out2** should always be the same):

```
αout1[γplate, αin, ϕin]
αout2[γplate, αin, ϕin]
0.320796

0.320796
```

Output ellipticity and overall phase (* the numbers in this case are similar because γplate = π/4, yielding 0 in the numerator of the expression for δ *). Note that the ellipticity is equal but reversed in direction.

```
γout[γplate, αin, ϕin]
θout[θin, γplate, αin, ϕin]
-0.1309

0.1309
```

The form of the output optical pulse given the above parameters would be,

$$\text{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = \text{Eout}\, e^{i\omega_o t}\, e^{i\theta\text{out}} \begin{pmatrix} \text{Cos}\,(\alpha\text{out}) \\ \text{Sin}\,(\alpha\text{out})\, e^{i\phi\text{out}} \end{pmatrix}$$

$$\text{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = 0.944061\ \text{Eo}\ e^{i\,\omega_o\,t}\, e^{i*0.1309} \begin{pmatrix} \text{Cos}[0.320796] \\ \text{Sin}[0.320796]\, e^{-i*0.1309} \end{pmatrix}$$

in other words,

AmplitudeOut	=	0.944061 * Eo
ωoOut	=	ωoIn
θOut	=	θIn+0.1309
αOut	=	0.320796
ϕOut	=	-0.1309

COTS Website notes:

http://www.nxtbook.com/nxtbooks/newportcorp/resource2011/#/800

http://demonstrations.wolfram.com/PolarizationOfLightThroughAWavePlate/ (* if you have mathematica installed, this demonstration project is excellent *)

http://www.icwic.com/icwic/data/pdf/cd/cd069/Polarizers,%20FO/a/111999.pdf
http://www.ozoptics.com/ALLNEW_PDF/DTS0072.pdf

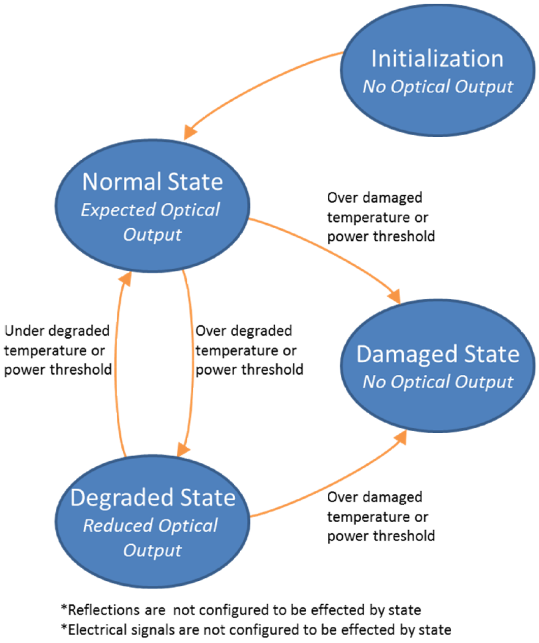
J.9 Component Use Case

J.9.1 Respond to an Optical Packet in the Half-wave Plate

Optical packet arrives at half-wave plate. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the half-wave plate. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated, changed optical output signal based on the input signal, component characteristics and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
  - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 93. Component states.

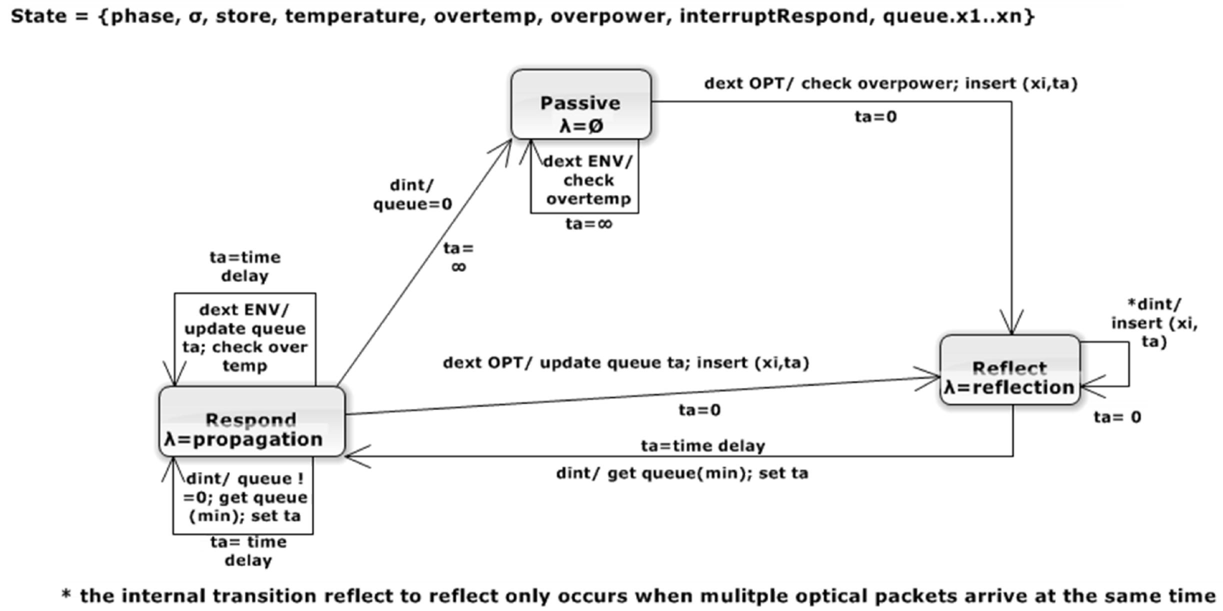

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta=∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
Respond
λ=propagation
ta=0
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 94. Half-wave plate phase transition diagram.

J.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Optical packet attenuated properly to the limit of accuracy
- Optical packet propagated out the correct port at the correct time

J.9.3 Respond to an Environmental Packet in the Half-wave Plate

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

J.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

J.10 Half-wave Plate Test Cases.

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Half-wave Plate Test Cases.*

		Inject Ports				Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Env**	Notes	opt #	**env #**
Passive	1	1	0	0	single	1	0
	2	0	1	0	single	2	0
	3	0	0	1	single	2	1
	4	1	1	0	same time	4	1
	5	1	1	0	differ time	6	1
	6	1	1	1	same time	8	2
	7	1	1	1	differ time	10	3
	8	0	1	1	same time	11	4
	9	0	1	1	differ time	12	5
	10	1	0	1	same time	13	6
	11	1	0	1	differ time	14	7
	20	2	0	0	same time	16	7
	21	0	2	0	same time	18	7
	22	2	2	0	same time	22	7
	23	2	2	0	differ time	26	7
	24	2	2	1	same time	30	8
	25	2	2	1	differ time	34	9
	26	0	2	1	same time	36	10
	27	0	2	1	differ time	38	11
	28	2	0	1	same time	40	12
	29	2	0	1	differ time	42	13
totals		21	21	13	42		
Respond	41	2	0	0	single	44	13
	42	0	2	0	single	46	13
	43	1	0	1	single	47	14
	44	2	1	0	same time	50	14
	45	2	1	0	differ time	53	14
	46	2	1	1	same time	56	15
	47	2	1	1	differ time	59	16
	48	0	2	1	same time	61	17
	49	0	2	1	differ time	63	18
	50	2	0	1	same time	65	19
	51	2	0	1	differ time	67	20

60	3	0	0	same time	70	20
61	0	3	0	same time	73	20
62	3	2	0	same time	78	20
63	3	2	0	differ time	83	20
64	3	2	1	same time	88	21
65	3	2	1	differ time	93	22
66	0	3	1	same time	96	23
67	0	3	1	differ time	99	24
68	3	0	1	same time	102	25
69	3	0	1	differ time	105	26
totals	36	27	13	63		
TC1	1	0	2	single	106	28
TC2	1	0	2	single	107	30
TC3	1	0	2	single	108	32
TC4	1	0	2	single	109	34
TC5	1	0	2	single	110	36
TC6	1	0	2	single	111	38
TC7	1	0	2	single	112	40
TC8	1	0	2	single	113	42
totals	8	0	16	24		

J.11 References

Newport. (2013). Polarization - wave plates. Retrieved September 20, 2013, from http://www.newport.com/Polarization/144921/1033/content.aspx

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

Appendix K - In-Line Polarizer

K.1 Device Description:

The in-line polarizer is a filter that allows light of a one polarization to pass while blocking light that is orthogonal to the passed light. It can convert unpolarized light into polarized light or filter out extraneous polarization angles from already polarized light. The type of polarizer used in QKD devices is the in-line fiber polarizer, which consists of housing containing input and output lenses with some form of polarizing medium in between. See Figure 1 for an example of an in-line polarizer.

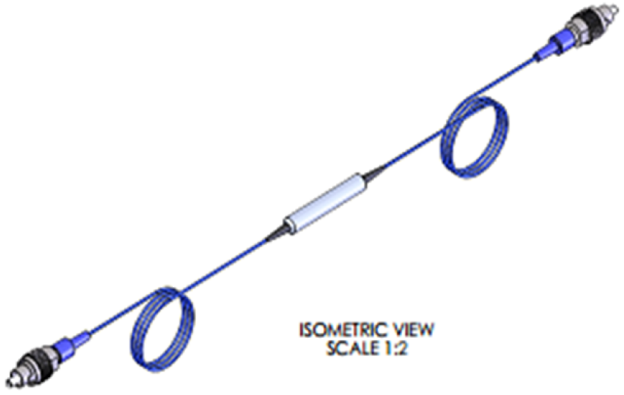


Figure 95. View of a fiber in-line polarizer (ThorLabs, 2013).

The In-line polarizer is a bidirectional optical component with two optical ports. Optical signals arriving at the input port are propagated to the other port after a defined propagation delay and the polarizing material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the In-line polarizer may become permanently damaged which changes its propagation characteristics. Similarly, the In-line polarizer is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the In-line polarizer may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the In-line polarizer is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the In-line polarizer. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the In-line polarizer to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the In-line polarizer using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the In-line polarizer. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

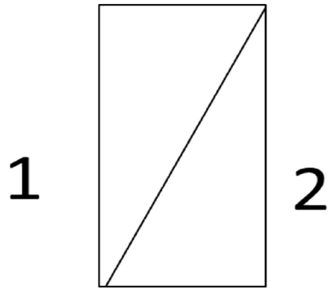


Figure 96. Symbol for the In-line polarizer in the QKD system architecture.

K.2 In-line polarizer Conceptual Model

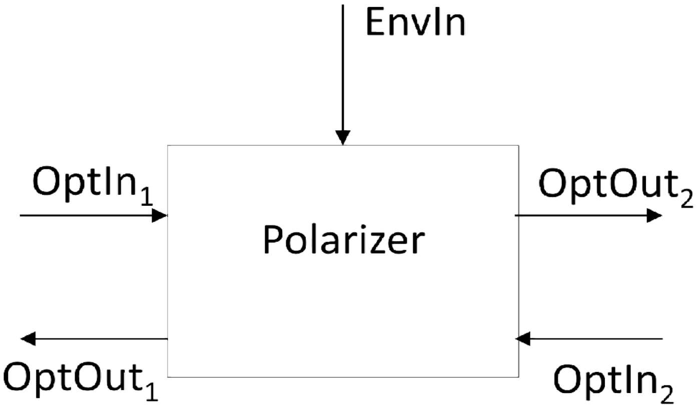


Figure 97. In-line polarizer conceptual model.

The conceptual model for an In-line polarizer consists of two optical input ports {OptIn_1, OptIn_2}, two optical output ports {OptOut_1, OptOut_2}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the In-line polarizer. In comparison to the In-line polarizer symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the In-line polarizer, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 3 will instantaneously generate a reflected emitting out of OptOut_1.

The In-line polarizer calculates changes to the power, the amplitude, ellipticity and polarization of any packet coming through either optical port after a time equaling the propagation delay of the module. The packet is calculated at full power minus some small amount to account for attenuation through the device but heavily attenuates any packet that does not match the polarization of the polarizer. Even though the in-line polarizer is meant to block light that does not match its polarization, a small amount of light that nearly matches the polarization of the polarizer will make it through the device and out the input port.

The In-line polarizer must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the In-line polarizer can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

K.3 Mathematical Model

For a detailed mathematical description of the In-line polarizer, refer to Section 9.8 which contains the Mathematica worksheet provided by the optical physics SME.

K.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the In-line polarizer.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Place the optical packet into the queue
- After the propagation time has elapsed, retrieve the input optical signal from the queue, and determine the input port and polarization of the signal.
- Update the values of the input optical signal based on the characteristics of the polarizer, the original values of the input optical signal and the current environment.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

K.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the In-line polarizer in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the In-line polarizer at the same time.

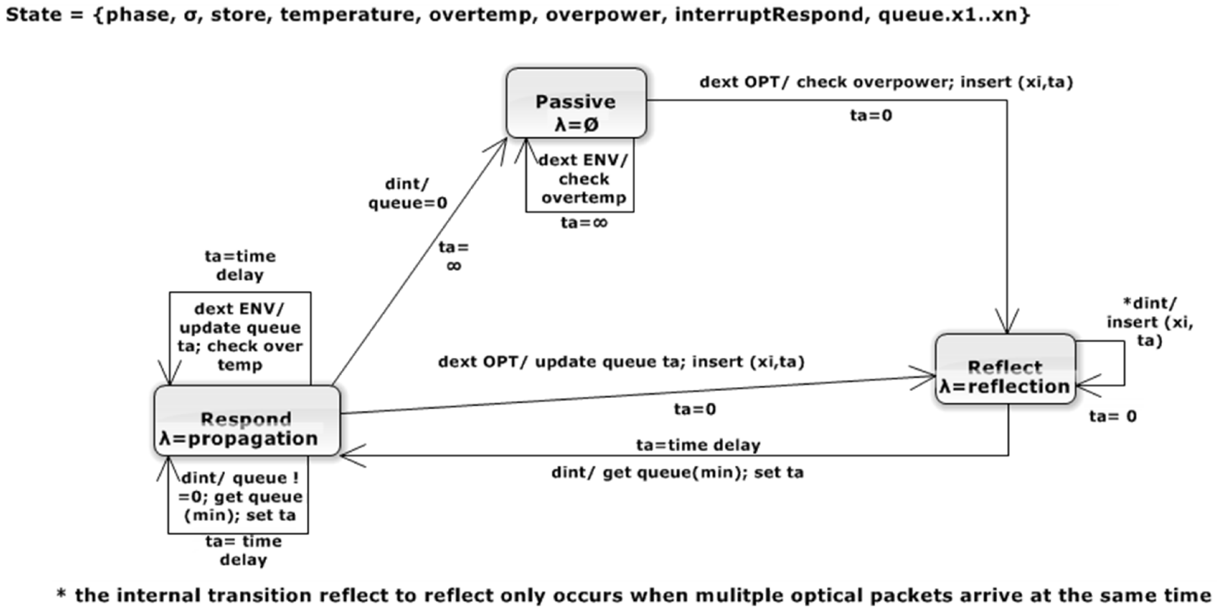

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta=∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
Reflect
λ=reflection
dext OPT/ update queue ta; insert (xi,ta)
ta=0
Respond
λ=propagation
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 98. In-line polarizer phase transition diagram.

K.6 Event-Trace Diagram

This section shows various examples of packets entering the FOA. The tables list the states the attenuator proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the attenuator, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

K.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 40. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	

5	s3	entry	passive	inf	x1	c	n	n	n	null	

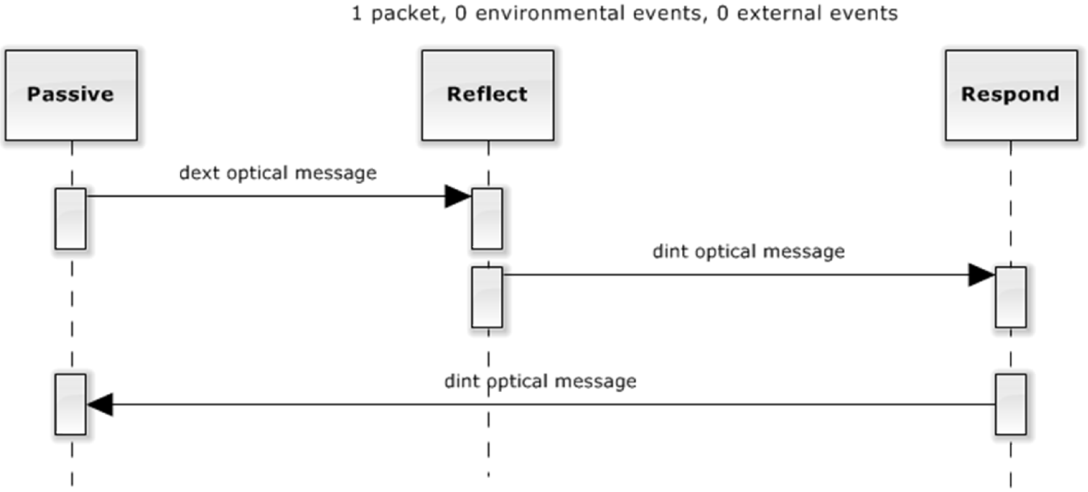


Figure 99. Case I sequence diagram.

K.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 41. *Case II state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**Interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

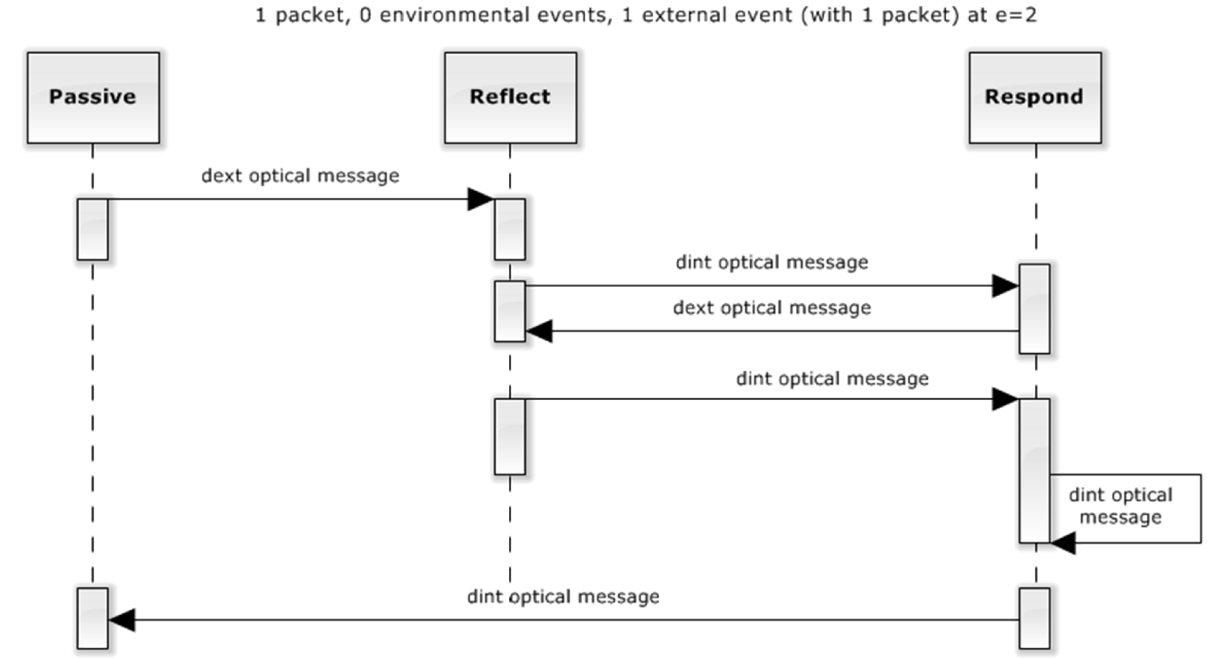


Figure 100. Case II sequence diagram.

K.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 42. *Case III state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	

3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

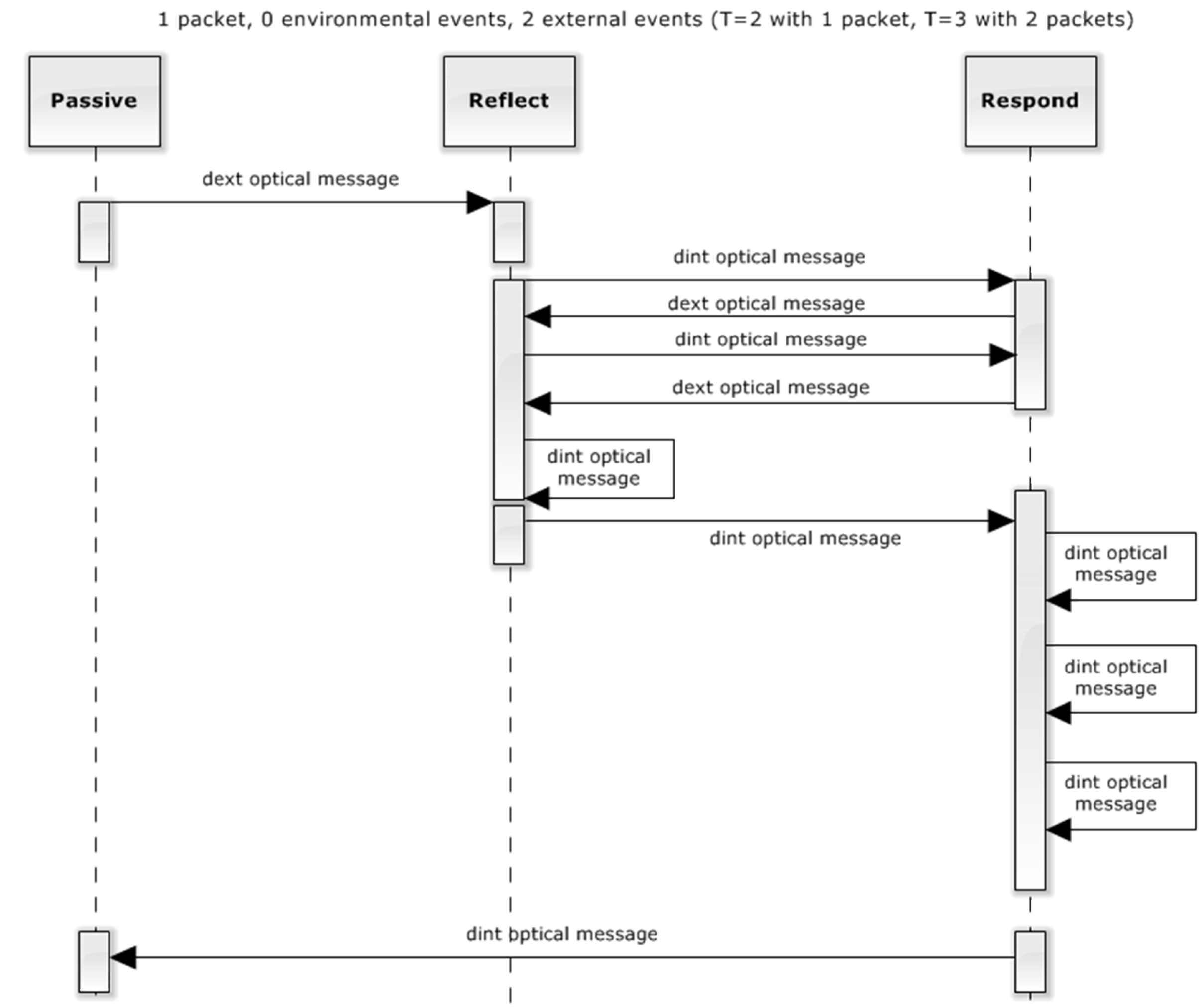


Figure 101. Case III sequence diagram.

K.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 43. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

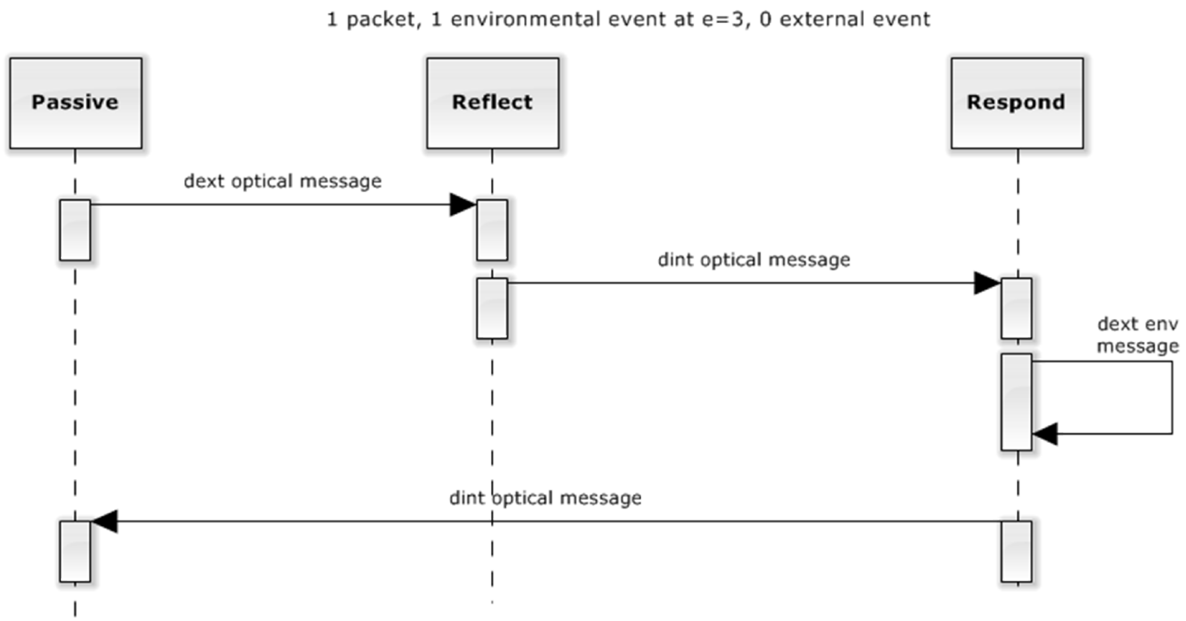


Figure 102. Case IV sequence diagram.

K.7 In-line polarizer Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.

- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, "store", temperature, "overtemp", "overpower", "interruptRespond", queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level
"overpower" = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the In-line polarizer we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \text{p} \in \mathit{InPorts}, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in \mathit{OutPorts}, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{\textit{In-line polarizer}}$* **= ($X_M$, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {"passive", "reflect", "respond"}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level

interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time delay_i) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay_i) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet

MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its x (p,v) values into the variable *store*

InPorts = {"OptIn$_1$", "OptIn$_2$", "EnvIn"} with
X_M = {("OptIn$_1$", V_{opt}), ("OptIn$_2$", V_{opt}), ("EnvIn", V_{env})} is the set of input ports and values.

OutPorts = {"OptOut$_1$", "OptOut$_2$"} with
Y_M = {("OptOut$_1$", $Y_{OptOut1}$), ("OptOut$_2$", $Y_{OptOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond, queue*} = {{"passive", "reflect", "respond"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x {"Y","N"} x *V*}

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue, e, (($p_i$,$v_i$),....
                                                                                              ($p_n$,$v_n$))) =
("reflect", 0, store, temperature, overtemp, overpower,interruptRespond, queue.x1..xn)
  if phase = "passive" and $p \in$ {"OptIn$_1$", "OptIn$_2$"}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower = "Y"
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = "N"

("reflect", 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = "respond" and $p \in$ {"OptIn$_1$", "OptIn$_2$"}
    update_delay(queue)
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower = "Y"
      insert_event_q(current)
       remove_event_m(current)
   queue.current = queue_need_reflected()
   reflect = (queue.current.p), calcReflected(queue.current.v))
```

```
    mark_reflected(queue.current)
    interruptRespond= “Y”
    timeLeftRespond = timeLeftRespond - e

 (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “passive” and p = “EnvIn”
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                      queue.x1..xn)
    if phase = “respond” and p = “EnvIn”
      update_delay(queue)
      timeLeftRespond = time_delay- e
      temperature = messagebag.temperature
      if temperature > damage.temp
        overtemp = “Y”
      time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
(“reflect”, 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
   if phase = “reflect” and need.reflect != null
     need.reflect = queue_need_reflected()
     current = need.reflect
    reflect = (current.p), calcReflected(current.v))
    mark_reflected(current)

 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
   if phase = “reflect” and need.reflect = null
     need.reflect = queue_need_reflected()
     if interruptRespond = “N”
       current = queue_min()
       time.delay = current.time.delay
       if InPort = “OptIn$_1$”
         outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
         outputPort = “OptOut$_2$”
       if InPort = “OptIn$_2$”
         outputPulse = calcAtten(current.v, temperature, overtemp, peakpwr, overpwr)
         outputPort = “OptOut$_1$”
```

```
      timeLeftRespond = propagation delay
    else
      time.delay = timeLeftRespond

  ("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "respond" and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if InPort = "OptIn$_1$"
        outputPulse = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "OptOut$_2$"
      if InPort = "OptIn$_2$"
        outputPulse = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "OptOut$_1$"
      interruptRespond= "N"

  ("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "respond" and size  = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

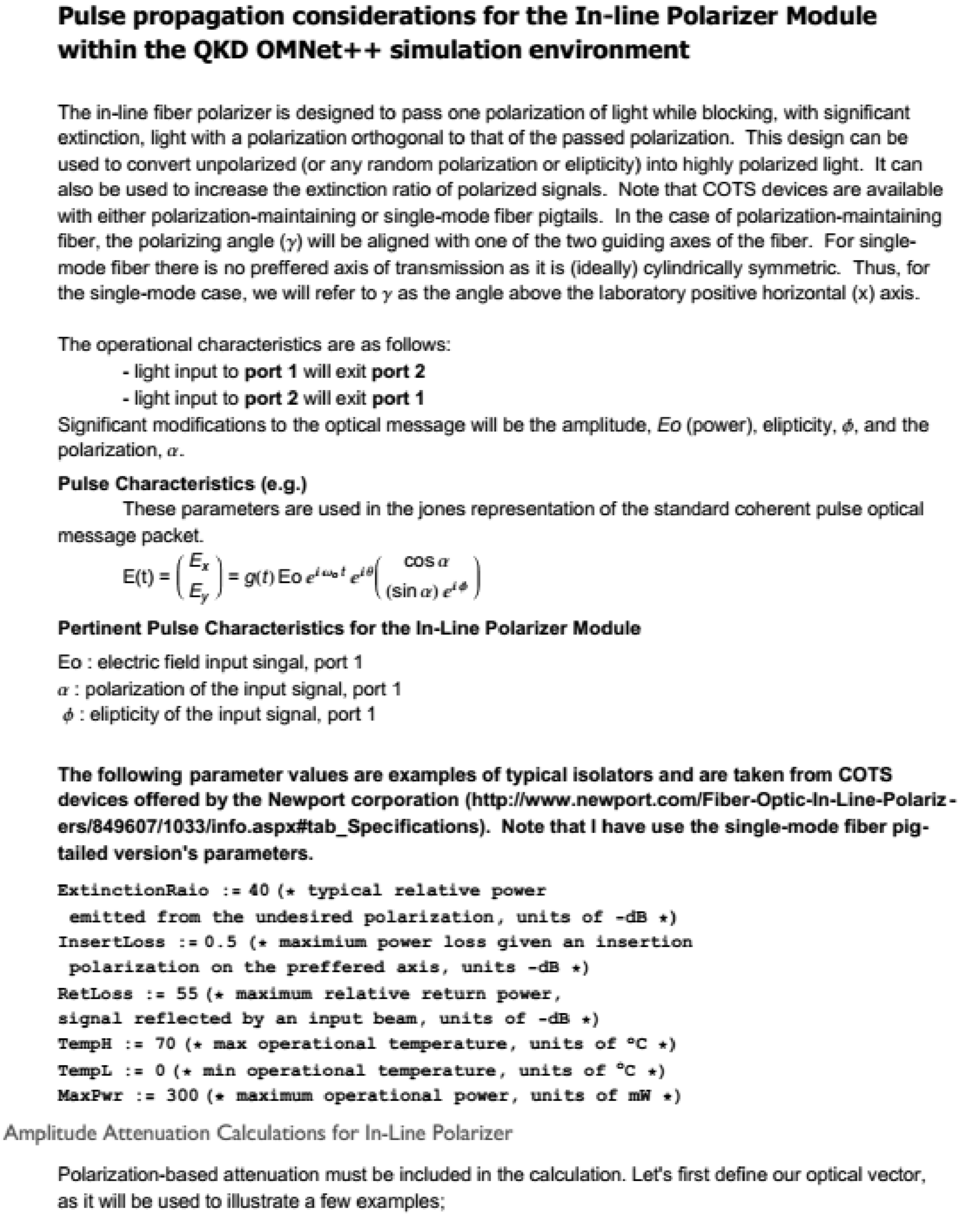
Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
  (reflect.p, reflect.v)
    if phase = "reflect"

  (output.port, output.pulse)
    if phase = "respond"

  $\emptyset$ (null output)
   otherwise;
```

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower, interruptRespond, queue) = \sigma$;

K.8 Appendix A – Mathematical model



Pulse propagation considerations for the In-line Polarizer Module within the QKD OMNet++ simulation environment

The in-line fiber polarizer is designed to pass one polarization of light while blocking, with significant extinction, light with a polarization orthogonal to that of the passed polarization. This design can be used to convert unpolarized (or any random polarization or elipticity) into highly polarized light. It can also be used to increase the extinction ratio of polarized signals. Note that COTS devices are available with either polarization-maintaining or single-mode fiber pigtails. In the case of polarization-maintaining fiber, the polarizing angle (γ) will be aligned with one of the two guiding axes of the fiber. For single-mode fiber there is no preffered axis of transmission as it is (ideally) cylindrically symmetric. Thus, for the single-mode case, we will refer to γ as the angle above the laboratory positive horizontal (x) axis.

The operational characteristics are as follows:

- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

Significant modifications to the optical message will be the amplitude, *Eo* (power), elipticity, ϕ, and the polarization, α.

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E}(t) = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\, e^{i\,\omega_0\,t}\, e^{i\,\theta} \begin{pmatrix} \cos\alpha \\ (\sin\alpha)\, e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the In-Line Polarizer Module

Eo : electric field input singal, port 1

α : polarization of the input signal, port 1

ϕ : elipticity of the input signal, port 1

The following parameter values are examples of typical isolators and are taken from COTS devices offered by the Newport corporation (http://www.newport.com/Fiber-Optic-In-Line-Polarizers/849607/1033/info.aspx#tab_Specifications). Note that I have use the single-mode fiber pigtailed version's parameters.

```
ExtinctionRaio := 40 (* typical relative power
 emitted from the undesired polarization, units of -dB *)
InsertLoss := 0.5 (* maximium power loss given an insertion
 polarization on the preffered axis, units -dB *)
RetLoss := 55 (* maximum relative return power,
signal reflected by an input beam, units of -dB *)
TempH := 70 (* max operational temperature, units of °C *)
TempL := 0 (* min operational temperature, units of °C *)
MaxPwr := 300 (* maximum operational power, units of mW *)
```

Amplitude Attenuation Calculations for In-Line Polarizer

Polarization-based attenuation must be included in the calculation. Let's first define our optical vector, as it will be used to illustrate a few examples;

$$\mathtt{OptVect := A\, e^{i\,Re[\omega o\, t]}} \begin{pmatrix} \mathtt{Cos[\alpha]} \\ \mathtt{Sin[\alpha]\, e^{i\phi}} \end{pmatrix}$$

Here, *A* is the input amplitude, α is the polarization, and ϕ is the ellipticity.

For the case of single-mode fiber pigtails we use the horizontal lab frame as γ=0 (vertical as γ=π/2), where γ can be any value [0 : π/2). For the case of polarization-maintaining fiber pigtails, γ will have the values of only 0 or π/2. In either case, the tranformation matrix is,

$$\mathtt{Polarizer[\gamma\_] :=} \begin{pmatrix} \mathtt{(Cos[\gamma])^2} & \mathtt{(Cos[\gamma] * Sin[\gamma])} \\ \mathtt{(Cos[\gamma] * Sin[\gamma])} & \mathtt{(Sin[\gamma])^2} \end{pmatrix}$$

To calculate the output amplitude we first want to account for the insertion loss

$$\mathtt{Etemp[Amplitude\_,\ InsertionLoss\_] := Amplitude * \sqrt{10^{-InsertionLoss/10}}}$$

Here we use the aforementioned insertion loss of -0.5 dB and the input amplitude Eo

```
EAfterInsertLoss = Etemp[A, InsertLoss]
0.944061 A
```

We now have to calculate the effective amplitude of the electric field after the light has passed through the polarizer. This is accomplished by taking the norm of the vector produced by operating the polarization matrix on the optical vector;

```
Eout[γ_] = Norm[Polarizer[γ].OptVect] // FullSimplify
```

$$\mathtt{Abs\left[Cos[\alpha]\ Cos[\gamma] + e^{i\phi}\ Sin[\alpha]\ Sin[\gamma]\right]\ \sqrt{A\ Conjugate[A]\ Cosh[2\ Im[\gamma]]}}$$

If we wish to flag the attenuator to include **undesired return (reflected)** messages, the following operations would hold true,

$$\mathtt{Eout[Ein\_,\ RetLoss\_] := Ein * \sqrt{10^{-RetLoss/10}}}$$

Examples For Amplitude Attenuation in the In-Line Polarizer

As an example, let's take the polarizer and input polarization to be oriented in the same, positive diagonal (45^o) with no elipticity;

$$\mathtt{Eout\left[\frac{\pi}{4}\right]\ /.\ \alpha \to \frac{\pi}{4}\ /.\ \phi \to 0\ //\ FullSimplify}$$

```
Abs[A]
```

As it should be, the only loss in the first example is due to insertion loss, which is not included here for illustrative purposes. To include insertion loss, simply define the amplitude *A* to be *EAfterInsertLoss*, i.e.,

```
Eout[π/4] /. α → π/4 /. ϕ → 0 /. A → EAfterInsertLoss // FullSimplify
0.944061 Abs[A]
```

As a second example, again excluding insertion loss, let' s keep the polarizer and input polarization at the same angle, but make the light circular ($\alpha = \pi/4$, $\phi = \pi/2$). Insertion loss is not included hereafter for the sake of clarity;

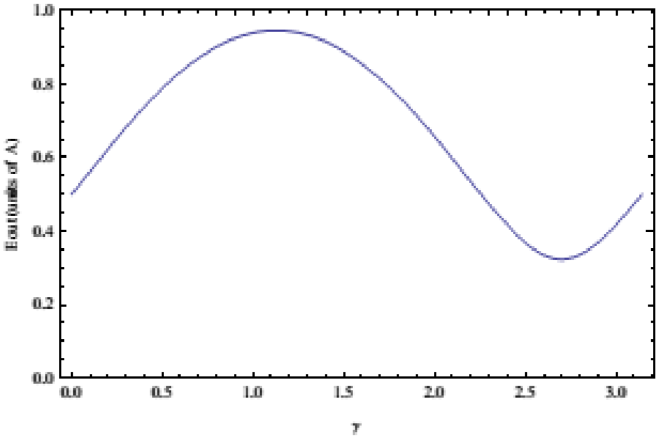
```
Eout[π/4] /. α → π/4 /. ϕ → π/2 // FullSimplify // N
0.707107 Abs[A]
```

Because the polarization "rotates", a power factor of 1/2 will be lost ($1/\sqrt{2}$ in the amplitude). Because the light is circular, the orientation of the polarizer won't effect the output amplitude, even if perpendicular to the initial input "polarization";

```
Eout[3 π/4] /. α → π/4 /. ϕ → π/2 // FullSimplify // N
0.707107 Abs[A]
```

The same doesn't hold true for elliptically polarized light. In the plot below, I change α to $\frac{\pi}{3}$, making the light slightly elliptical. I vary the polarizer angle γ from 0 to π;

```
Plot[Eout[γ] /. α → π/3 /. ϕ → π/4 /. A → 1, {γ, 0, π}, Frame → True,
 PlotRange → {0, 1}, FrameLabel → {"γ", "Eout (units of A)"}]
```



This plot is fun to play with. Note what happens when you have circular light ($\alpha=\pi/4$, $\phi=\pi/2$), linear light (α = any angle, $\phi = 0$) or any elliptical light (vary α or ϕ).

As a final example, let's put the polarizer angle at horizontal ($\gamma = 0$), but change the input polarization to vertical, with no ellipticity

```
Eout[0] /. α → π/2 /. ϕ → 0 // FullSimplify
0
```

This is idealized, as the devices typically call for an extinction ratios of -30 to -40 dB.

Polarizaion Calculations for In-Line Polarizer

In all cases, the light exiting the polarizer will be nearly pure (within the tolerances set by the extinction ratio) in the axis, γ, of the polarizer. Thus, for all cases,

$$\textbf{OutputPol} := \gamma$$

Final Example for an optical pulse passed through the In-Line Polarizer (refresh Kernel before use)

$$\textbf{InsertLoss} := 0.5$$

$$\textbf{OptVect}[\alpha\_, \phi\_] := A\, e^{i\,\text{Re}[\omega o\, t]} \begin{pmatrix} \text{Cos}[\alpha] \\ \text{Sin}[\alpha]\, e^{i\,\phi} \end{pmatrix}$$

$$\textbf{Polarizer}[\gamma\_] := \begin{pmatrix} (\text{Cos}[\gamma])^2 & (\text{Cos}[\gamma] * \text{Sin}[\gamma]) \\ (\text{Cos}[\gamma] * \text{Sin}[\gamma]) & (\text{Sin}[\gamma])^2 \end{pmatrix}$$

$$\textbf{Eout}[\gamma\_, \alpha\_, \phi\_] = \sqrt{10^{-\text{InsertLoss}/10}} * \textbf{Norm}[\textbf{Polarizer}[\gamma].\textbf{OptVect}[\alpha, \phi]] \;//\; \textbf{FullSimplify}$$

$$0.944061\, \text{Abs}\left[\text{Cos}[\alpha]\, \text{Cos}[\gamma] + e^{i\,\phi}\, \text{Sin}[\alpha]\, \text{Sin}[\gamma]\right] \sqrt{A\, \text{Conjugate}[A]\, \text{Cosh}[2\, \text{Im}[\gamma]]}$$

In the example below, I have assigned the polarizer angle to be just above horizontal ($\gamma = \frac{\pi}{16}$) with the light polarized at positive diagonal ($\alpha = \pi/4$) with a slight ellipticity ($\phi = \pi/32$)

$$\textbf{OutputAmplitude} = \textbf{Eout}\left[\frac{\pi}{16}, \frac{\pi}{4}, \frac{\pi}{32}\right] \;//\; \textbf{FullSimplify}$$

$$0.784435\, \text{Abs}[A]$$

$$\textbf{OutputPolariztion} = \frac{\pi}{16}$$

$$\frac{\pi}{16}$$

$$\textbf{OutputEllipticity} = 0$$

$$0$$

The form of the output optical pulse given the above parameters would be,

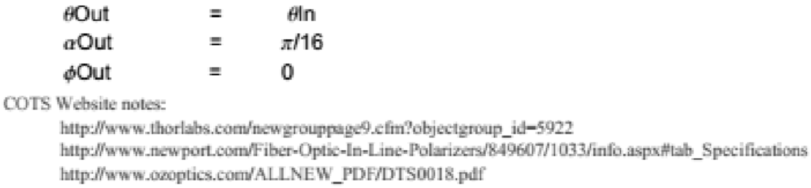
$$\text{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = \text{OutpuAmplitude}\, e^{i\omega_o t}\, e^{i\theta} \begin{pmatrix} \text{Cos (OuputPolarization)} \\ \text{Sin (OutputPolarization)}\, e^{i\,*\text{OutputEllipticity}} \end{pmatrix}$$

$$\text{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = 0.784435{*}A{*}(\, e^{i\,\omega_o\, t}\, e^{i\,\theta}) \begin{pmatrix} \text{Cos}[\pi/16] \\ \text{Sin}[\pi/16]\, e^{i*0} \end{pmatrix}$$

$$\text{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = 0.784435{*}A{*}(\, e^{i\,\omega_o\, t}\, e^{i\,\theta}) \begin{pmatrix} 0.980785 \\ 0.19509 \end{pmatrix}$$

in other words,

AmplitudeOut = 0.784435 * AmplitudeIn
ωoOut = ωoIn



θOut = θIn
αOut = $\pi/16$
ϕOut = 0

COTS Website notes:

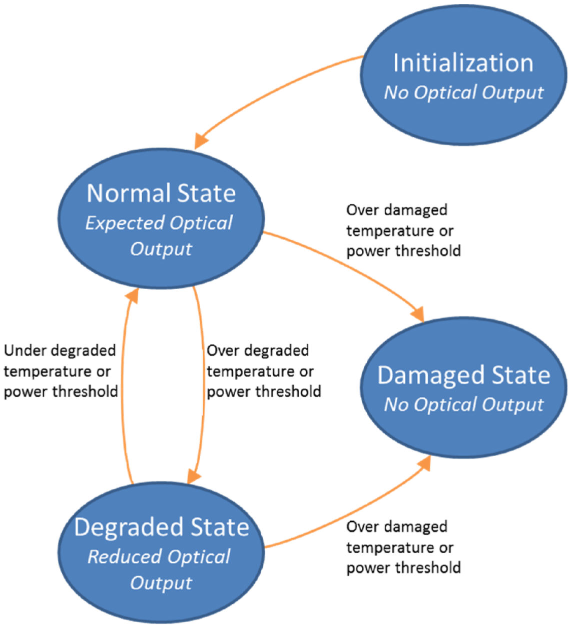
http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=5922
http://www.newport.com/Fiber-Optic-In-Line-Polarizers/849607/1033/info.aspx#tab_Specifications
http://www.ozoptics.com/ALLNEW_PDF/DTS0018.pdf

K.9 Component Use Case

K.9.1 Respond to an Optical Packet in the Inline Polarizer

Optical packet arrives at the inline polarizer. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the inline polarizer. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated and polarized optical output signal based on the input signal, component characteristics and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

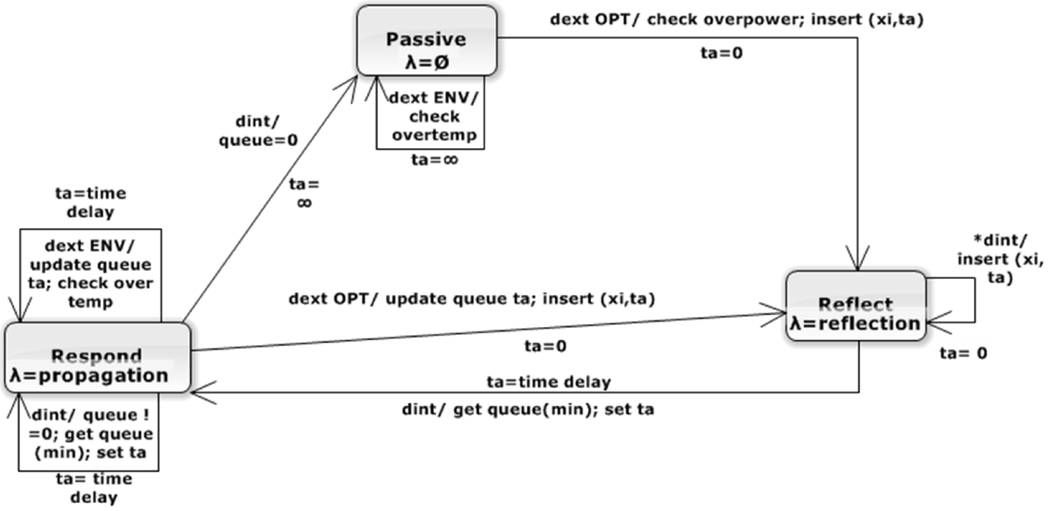
- Identified Alternative Uses Cases
    - React to an environmental message
- Assumptions
    - Component has completed initialization sequence at least once
    - Reflections are not affected by component state
    - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 103. Component states.

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}


Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
ta= 0
Respond
λ=propagation
ta=0
ta=time delay
dint/ get queue(min); set ta
dint/ queue != 0; get queue (min); set ta
ta= time delay

* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 104. In-line polarizer phase transition diagram.

K.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Optical packet attenuated properly to the limit of accuracy

- Optical packet propagated out the correct port at the correct time

K.9.3 Respond to an Environmental Packet in the In-line Polarizer

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

K.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

K.10 Inline Polarizer Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change

in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Inline Polarizer Test Cases.*

		Inject Ports				Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Env**	**Notes**	**opt #**	**env #**
Passive	1	1	0	0	single	1	0
	2	0	1	0	single	2	0
	3	0	0	1	single	2	1
	4	1	1	0	same time	4	1
	5	1	1	0	differ time	6	1
	6	1	1	1	same time	8	2
	7	1	1	1	differ time	10	3
	8	0	1	1	same time	11	4
	9	0	1	1	differ time	12	5
	10	1	0	1	same time	13	6
	11	1	0	1	differ time	14	7
	20	2	0	0	same time	16	7
	21	0	2	0	same time	18	7
	22	2	2	0	same time	22	7
	23	2	2	0	differ time	26	7
	24	2	2	1	same time	30	8
	25	2	2	1	differ time	34	9
	26	0	2	1	same time	36	10
	27	0	2	1	differ time	38	11

	28	2	0	1	same time	40	12
	29	2	0	1	differ time	42	13
totals		21	21	13	42		
Respond	41	2	0	0	single	44	13
	42	0	2	0	single	46	13
	43	1	0	1	single	47	14
	44	2	1	0	same time	50	14
	45	2	1	0	differ time	53	14
	46	2	1	1	same time	56	15
	47	2	1	1	differ time	59	16
	48	0	2	1	same time	61	17
	49	0	2	1	differ time	63	18
	50	2	0	1	same time	65	19
	51	2	0	1	differ time	67	20
	60	3	0	0	same time	70	20
	61	0	3	0	same time	73	20
	62	3	2	0	same time	78	20
	63	3	2	0	differ time	83	20
	64	3	2	1	same time	88	21
	65	3	2	1	differ time	93	22
	66	0	3	1	same time	96	23
	67	0	3	1	differ time	99	24
	68	3	0	1	same time	102	25
	69	3	0	1	differ time	105	26
totals		36	27	13	63		
	TC1	1	0	2	single	106	28
	TC2	1	0	2	single	107	30
	TC3	1	0	2	single	108	32
	TC4	1	0	2	single	109	34
	TC5	1	0	2	single	110	36
	TC6	1	0	2	single	111	38
	TC7	1	0	2	single	112	40
totals		7	0	14	21		

K.11 References

ThorLabs. (2013). In-line fiber optic polarizers .Retrieved September 18, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=5922

Appendix L - Isolator

L.1 Device Description

An isolator is a basic device used in fiber optic systems. The isolator is a device which is used to pass light in the forward direction while highly attenuating light moving in the opposite direction. This has the effect of operating as a “one-way street” and prevents reflected light from returning to a light source, such as a laser. This backward propagation of reflected light can have negative effects on the source. The attenuation is a fixed amount, usually expressed in decibels (dBs).

There are two types of isolators, the *polarization-dependent* and the *polarization-independent*. The dependent version of the isolator is usually constructed with an input polarizer, a Faraday rotator with a fixed magnet and an output polarizer. The input polarizer filters the incoming light, allowing only the parallel electric field components of an incoming beam to pass. The Faraday rotator rotates the light by 45° then outputs it to the second linear polarizer. The output light is aligned 45° to the incoming light. In the reverse direction the incoming light is polarized to 45°, and then passes through the Faraday rotator for another 45° rotation, meaning the light is now polarized perpendicular to the input polarizer and the light is either reflected or absorbed (Saleh & Teich, 1991). See Figure 1.

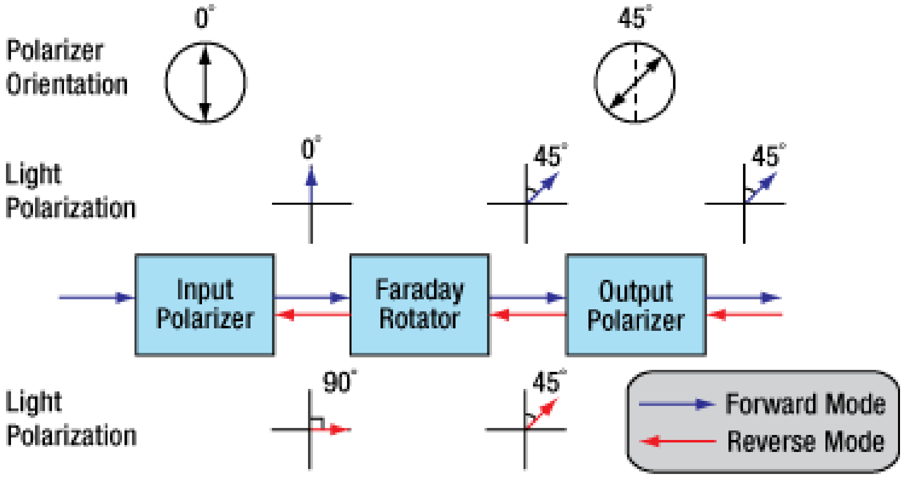


Figure 105. A polarization-dependent isolator (ThorLabs, 2013).

The independent version is somewhat more complicated as the light is split into to two streams by a birefringent crystal, then passes through a Faraday rotator which rotates the light by 45°, a half-wave plate then rotates the light by 45° again and finally through another birefringent crystal that recombines the beams into the output port. Reflected light passing into the output port passes through the birefringent crystal and split, then through the half-wave plate and Faraday rotator. When the light encounters the second birefringent crystal, the light is channeled away from the input port into the isolator housing (Saleh & Teich, 1991). See Figure 2. Both types of isolator may be used in a QKD device.

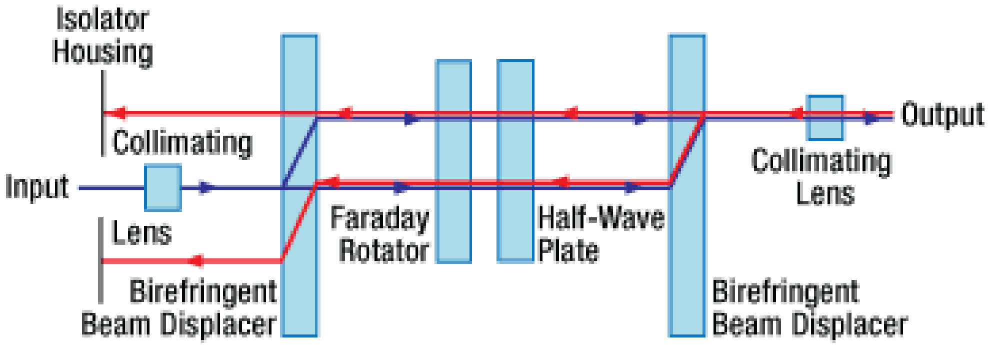


Figure 106. Polarization-independent isolator (ThorLabs, 2013).

The Isolator is a bidirectional optical component with one optical port. Optical signals arriving at the input port are propagated to the other port after a defined propagation delay.

Signals arriving at the output port are blocked from propagating through the input port. The Isolator is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the Isolator may become permanently damaged which changes its propagation characteristics. Similarly, the Isolator is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the Isolator may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the Isolator is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the Isolator. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the Isolator to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the Isolator using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the Isolator. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica

worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

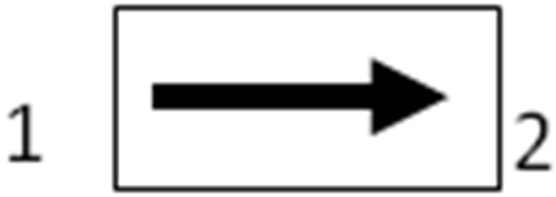


Figure 107. Symbol for the Isolator in the QKD system architecture.

L.2 Isolator Conceptual Model

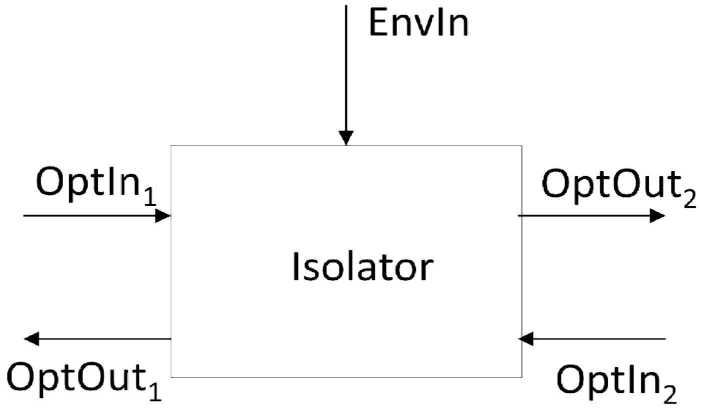


Figure 108. Isolator conceptual model.

The conceptual model for an Isolator consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the Isolator. In comparison to the Isolator symbol used in the QKD simulation architecture shown in Figure 3, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the Isolator, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 4 will instantaneously generate a reflected emitting out of OptOut_1.

The Isolator calculates the power of the incoming packet and if the packet comes in the normal input port, it is sent out the output port after a time equaling the propagation delay of the module at full power minus some small amount to account for attenuation through the device. If an incoming packet enters through the device output port, the packet is output through the input port after the propagation delay, but the packet power is heavily attenuated. Even though the isolator is meant to block light travelling in the wrong direction, a highly attenuated portion of backward propagating light will still pass through the device.

The Isolator must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the Isolator can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

L.3 Mathematical Model

For a detailed mathematical description of the Isolator, refer to Section 10.8 which contains the Mathematica worksheet provided by the optical physics SME.

L.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the Isolator.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the signal into a queue until the propagation time through the component has elapsed.

- After the propagation time has elapsed, retrieve the input optical signal from the queue, and determine the input port of the signal.
- If the input port of the signal matches the input port of the isolator, slightly attenuate the output signal with power based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment.
- If the input port of the signal does not match the input port of the isolator, heavily attenuate the output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

L.5 Phase Transition Diagram

The phase transition diagram in Fig. 5 shows the phases of the Isolator in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the Isolator at the same time.

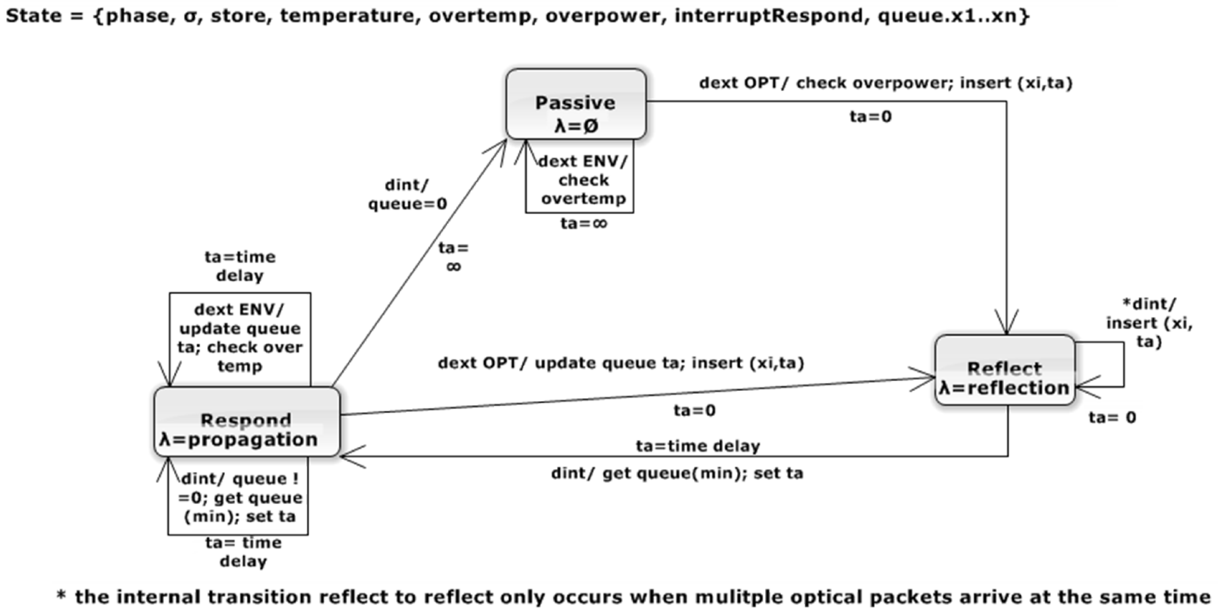

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
Reflect
λ=reflection
dext OPT/ update queue ta; insert (xi,ta)
ta=0
ta= 0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 109. Isolator phase transition diagram.

L.6 Event-Trace Diagram

This section shows various examples of packets entering the Isolator. The tables list the states the Isolator proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the Isolator, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable

- Queue: contents of the queue for that state
- Notes: any notes for that state

L.6.1 *CASE I: Initial Passive with Single Optical Packet Arriving at Time 0*

Table 44. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	
5	s3	entry	passive	inf	x1	c	n	n	n	null	

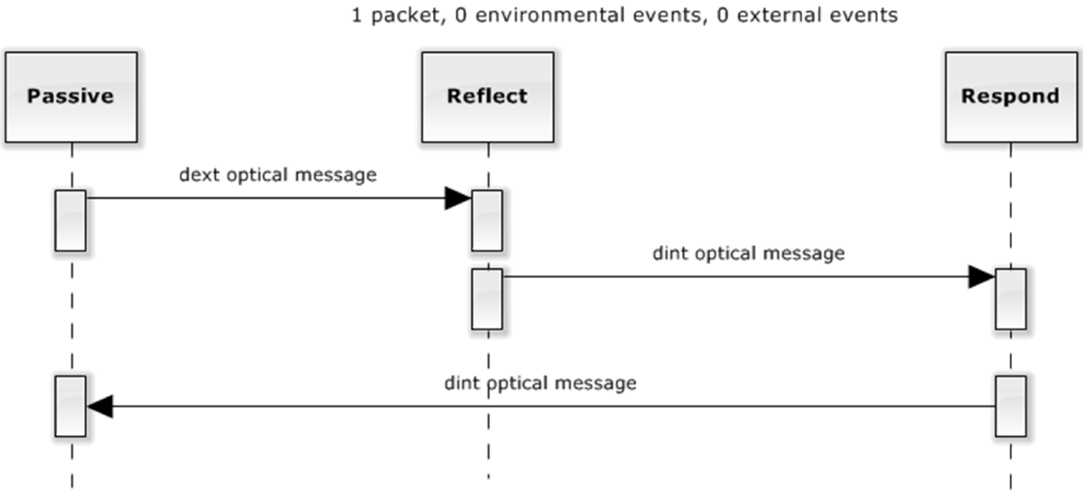


Figure 110. Case I sequence diagram.

L.6.2 *CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2*

Table 45. *Case II state list.*

											Notes:
		entry/			store		over	over	Interrupt	queue	assume
time	state	exit	phase	sigma	(*xi*)	temp	temp	power	respond	(*xi*, *tp*)	tp=5
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

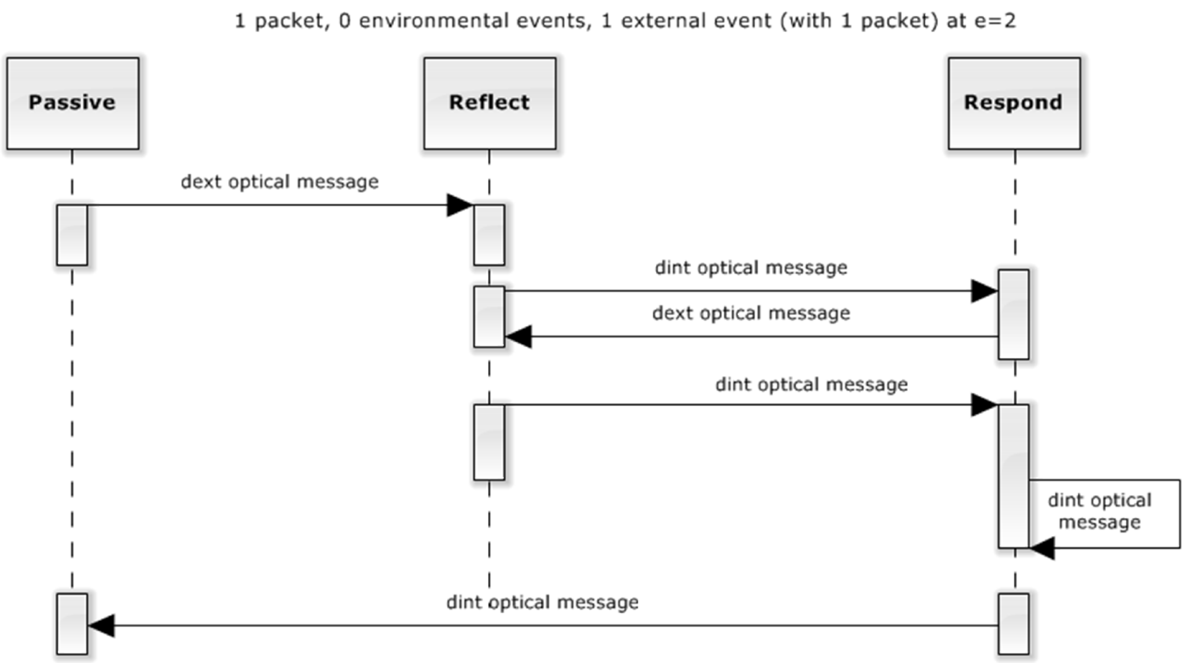


Figure 111. Case II sequence diagram.

L.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 46. *Case III state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	

8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

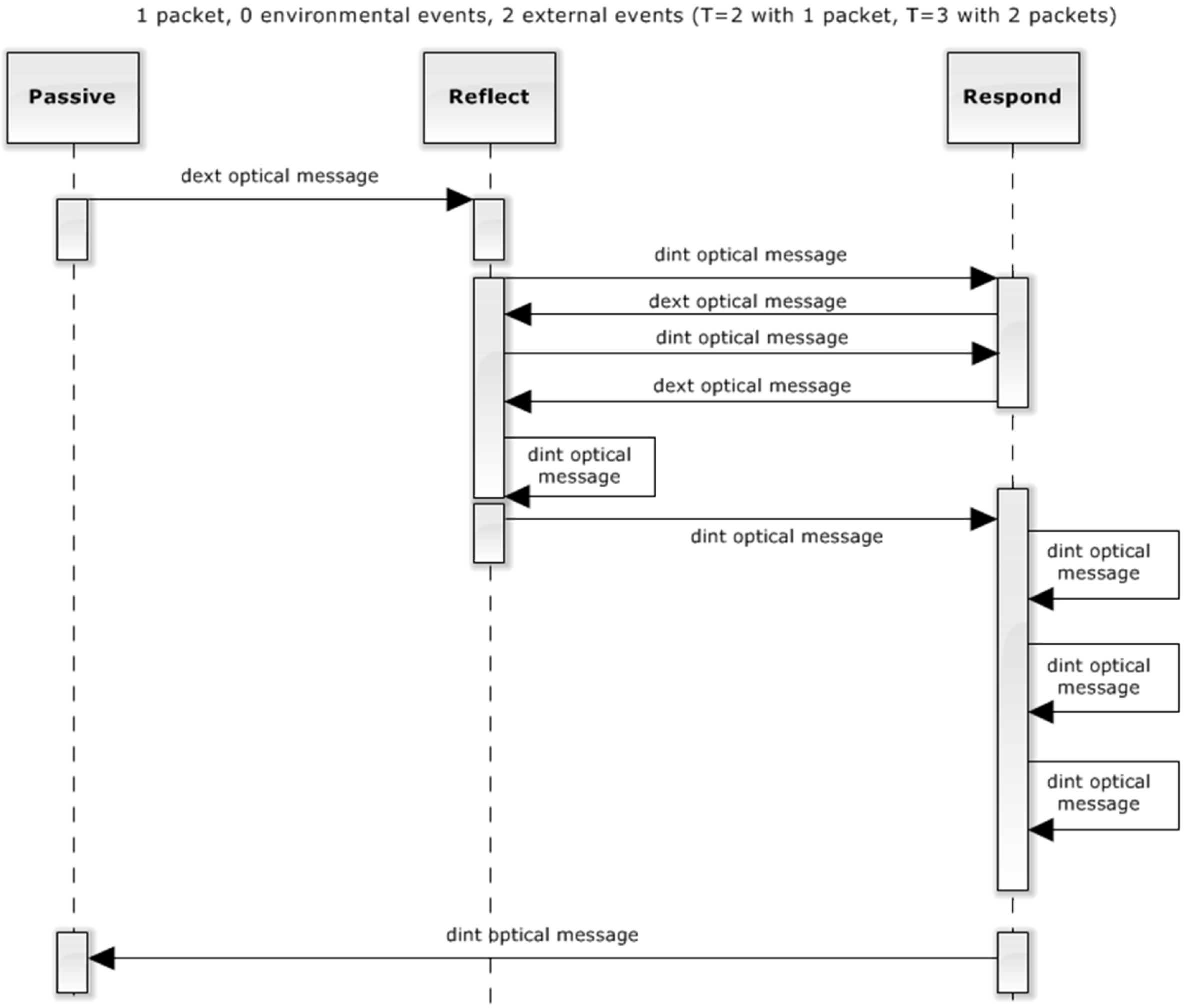


Figure 112. Case III sequence diagram.

L.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 47. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	

0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n	n	null	
5	s4	entry	passive	inf	x1	c2	y	n	n	null	

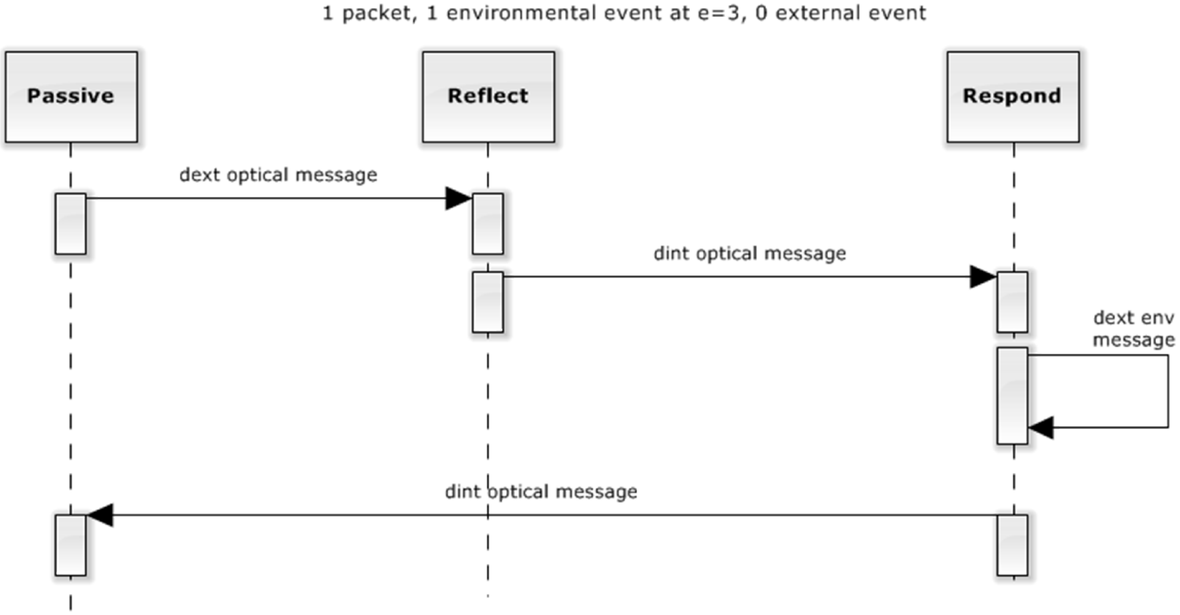


Figure 113. Case IV sequence diagram.

L.7 Isolator Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.

- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the Isolator we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \text{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS*$_{Isolator}$ = (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator

phase = {“passive”, “reflect”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time $delay_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcPolar() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“OptIn$_1$”, “OptIn$_2$”, “EnvIn”} with
X_M = {(“OptIn$_1$”, V_{opt}), (“OptIn$_2$”, V_{opt}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“OptOut$_1$”, “OptOut$_2$”} with
Y_M = {(“OptOut$_1$”, $Y_{OptOut1}$), (“OptOut$_2$”, $Y_{OptOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond*, *queue*} = {{“passive”, “reflect”, “respond”} x R_0^+ x V x R x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x V}

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
(“reflect”, 0, store, temperature, overtemp, overpower,interruptRespond,  queue.x1..xn)
  if phase = “passive” and $p \in$ {“OptIn$_1$”, “OptIn$_2$”}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower =  “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = “respond” and $p \in$ {“OptIn$_1$”, “OptIn$_2$”}
    update_delay(queue)
    for messagebag != null
     current = messagebag_first()
     if current.value.power > damaged.power
       overpower =  “Y”
     insert_event_q(current)
      remove_event_m(current)
    queue.current = queue_need_reflected()
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
   interruptRespond= "Y"
  timeLeftRespond = timeLeftRespond - e

("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = "passive" and p = "EnvIn"
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = "Y"

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = "respond" and p = "EnvIn"
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"
     time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
 otherwise;
```

Internal Transition Function:

δ_{int}(*phase*, σ, *store*, *temperature*, *overtemp*, *overpower*, *interruptRespond*, *queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =

```
("reflect", 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
  if phase = "reflect" and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = "reflect" and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = "N"
      current = queue_min()
      time.delay = current.time.delay
      if InPort = "$\text{OptIn}_1$"
        outputPulse = calcForward(current.v, temperature, overtemp, peakpwr, overpwr)
      outputPort = “OptOut$_2$”
    if InPort = “OptIn$_2$”
      outputPulse = calcReverse(current.v, temperature, overtemp, peakpwr, overpwr)
      outputPort = “OptOut$_1$”
  timeLeftRespond = propagation delay
 else
  time.delay = timeLeftRespond

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                  queue.x1..xn)
 if phase = “respond” and size > 0
  update_delay(queue)
  size= queue_size()
  current = queue_min()
  time.delay = current.time.delay
  if InPort = “OptIn$_1$”
    outputPulse = calcForward(current.v, temperature, overtemp, peakpwr, overpwr)
    outputPort = “OptOut$_2$”
  if InPort = “OptIn$_2$”
    outputPulse = calcReverse(current.v, temperature, overtemp, peakpwr, overpwr)
    outputPort = “OptOut$_1$”
  interruptRespond= “N”

(“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
 if phase = “respond” and size = 0
  size= queue_size()
```

Confluence Function:

$$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$$

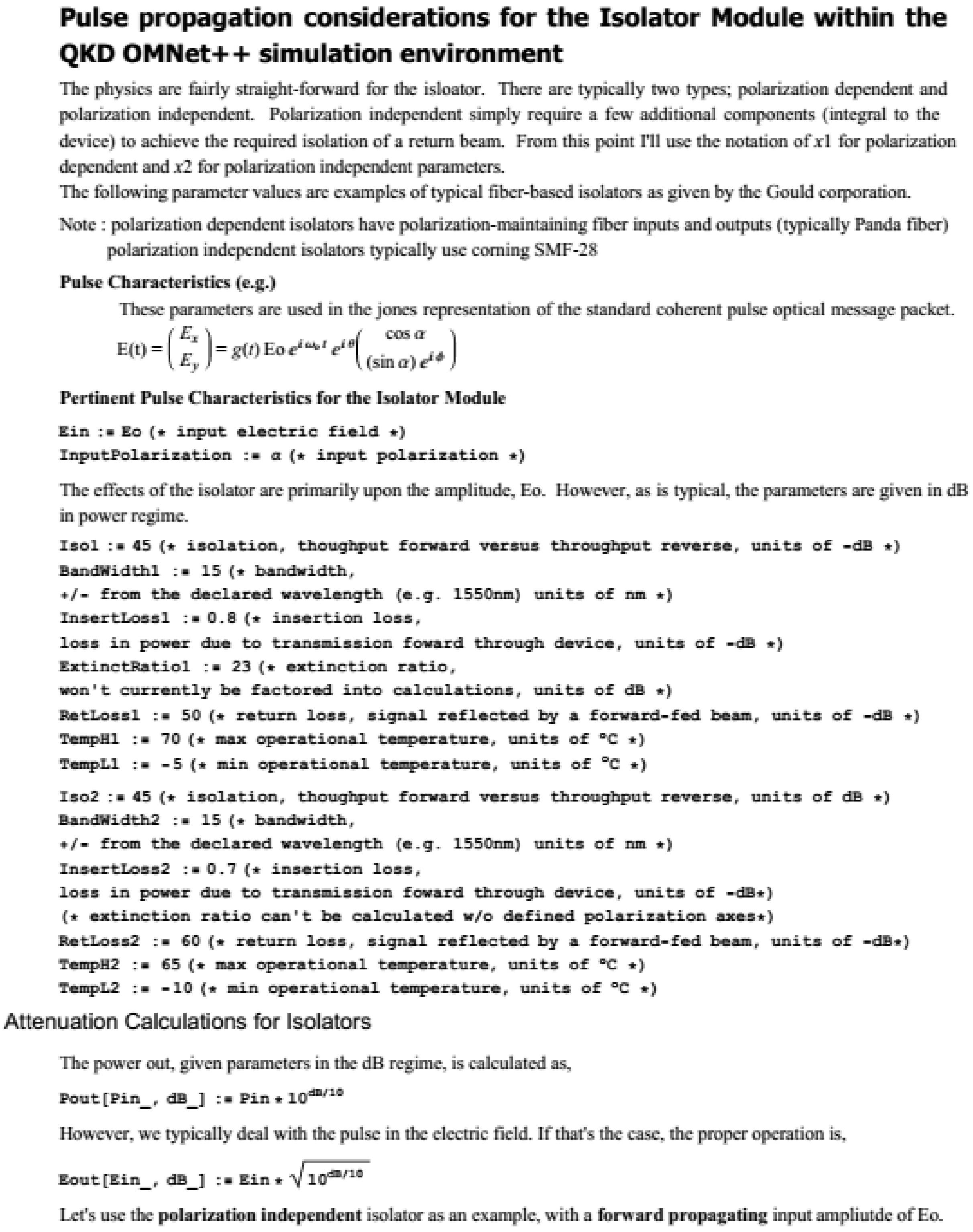
Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (outputPort, outputPulse)
    if phase = “propagate”

  ∅ (null output)
   otherwise;
```

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower, interruptRespond, queue) = \sigma;$

L.8 Mathematical Model



Pulse propagation considerations for the Isolator Module within the QKD OMNet++ simulation environment

The physics are fairly straight-forward for the isloator. There are typically two types; polarization dependent and polarization independent. Polarization independent simply require a few additional components (integral to the device) to achieve the required isolation of a return beam. From this point I'll use the notation of *x*1 for polarization dependent and *x*2 for polarization independent parameters.
The following parameter values are examples of typical fiber-based isolators as given by the Gould corporation.

Note : polarization dependent isolators have polarization-maintaining fiber inputs and outputs (typically Panda fiber)
polarization independent isolators typically use corning SMF-28

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\,\omega_o\,t}\,e^{i\,\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Isolator Module

```
Ein := Eo (* input electric field *)
InputPolarization := α (* input polarization *)
```

The effects of the isolator are primarily upon the amplitude, Eo. However, as is typical, the parameters are given in dB in power regime.

```
Iso1 := 45 (* isolation, thoughput forward versus throughput reverse, units of -dB *)
BandWidth1 := 15 (* bandwidth,
+/- from the declared wavelength (e.g. 1550nm) units of nm *)
InsertLoss1 := 0.8 (* insertion loss,
loss in power due to transmission foward through device, units of -dB *)
ExtinctRatio1 := 23 (* extinction ratio,
won't currently be factored into calculations, units of dB *)
RetLoss1 := 50 (* return loss, signal reflected by a forward-fed beam, units of -dB *)
TempH1 := 70 (* max operational temperature, units of °C *)
TempL1 := -5 (* min operational temperature, units of °C *)
```

```
Iso2 := 45 (* isolation, thoughput forward versus throughput reverse, units of dB *)
BandWidth2 := 15 (* bandwidth,
+/- from the declared wavelength (e.g. 1550nm) units of nm *)
InsertLoss2 := 0.7 (* insertion loss,
loss in power due to transmission foward through device, units of -dB*)
(* extinction ratio can't be calculated w/o defined polarization axes*)
RetLoss2 := 60 (* return loss, signal reflected by a forward-fed beam, units of -dB*)
TempH2 := 65 (* max operational temperature, units of °C *)
TempL2 := -10 (* min operational temperature, units of °C *)
```

Attenuation Calculations for Isolators

The power out, given parameters in the dB regime, is calculated as,

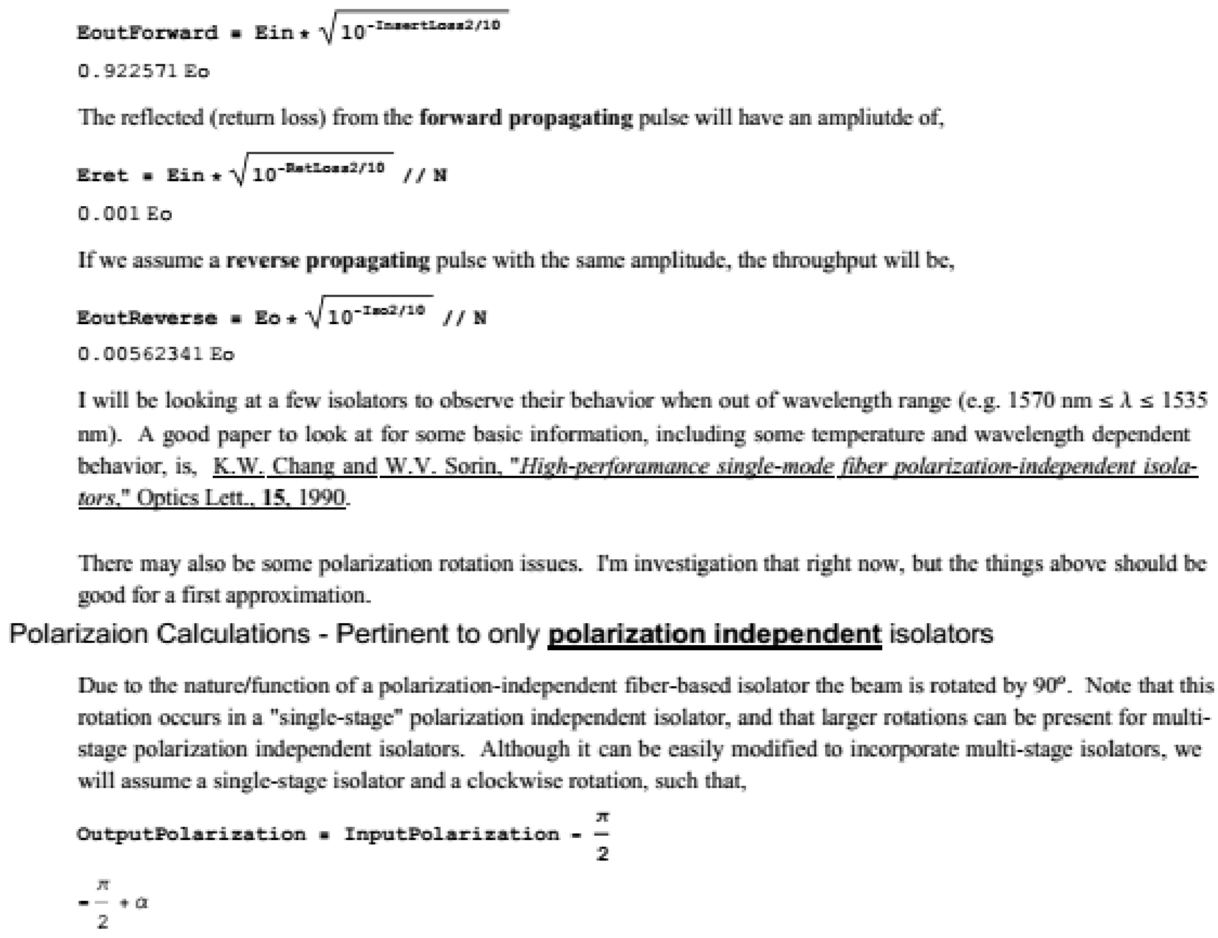
```
Pout[Pin_, dB_] := Pin * 10^(dB/10)
```

$$\mathrm{Pout[Pin\_,\ dB\_]} := \mathrm{Pin} * 10^{\mathrm{dB}/10}$$

However, we typically deal with the pulse in the electric field. If that's the case, the proper operation is,

$$\mathrm{Eout[Ein\_,\ dB\_]} := \mathrm{Ein} * \sqrt{10^{\mathrm{dB}/10}}$$

Let's use the **polarization independent** isolator as an example, with a **forward propagating** input ampliutde of Eo.



$$\text{EoutForward} = \text{Ein} * \sqrt{10^{-\text{InsertLoss2}/10}}$$

0.922571 Eo

The reflected (return loss) from the **forward propagating** pulse will have an ampliutde of,

$$\text{Eret} = \text{Ein} * \sqrt{10^{-\text{RetLoss2}/10}} \text{ // N}$$

0.001 Eo

If we assume a **reverse propagating** pulse with the same amplitude, the throughput will be,

$$\text{EoutReverse} = \text{Eo} * \sqrt{10^{-\text{Iso2}/10}} \text{ // N}$$

0.00562341 Eo

I will be looking at a few isolators to observe their behavior when out of wavelength range (e.g. 1570 nm $\leq \lambda \leq$ 1535 nm). A good paper to look at for some basic information, including some temperature and wavelength dependent behavior, is, K.W. Chang and W.V. Sorin, "*High-perforamance single-mode fiber polarization-independent isolators,*" Optics Lett., **15**, 1990.

There may also be some polarization rotation issues. I'm investigation that right now, but the things above should be good for a first approximation.

Polarizaion Calculations - Pertinent to only **polarization independent** isolators

Due to the nature/function of a polarization-independent fiber-based isolator the beam is rotated by 90°. Note that this rotation occurs in a "single-stage" polarization independent isolator, and that larger rotations can be present for multi-stage polarization independent isolators. Although it can be easily modified to incorporate multi-stage isolators, we will assume a single-stage isolator and a clockwise rotation, such that,

$$\text{OutputPolarization} = \text{InputPolarization} - \frac{\pi}{2}$$

$$-\frac{\pi}{2} + \alpha$$

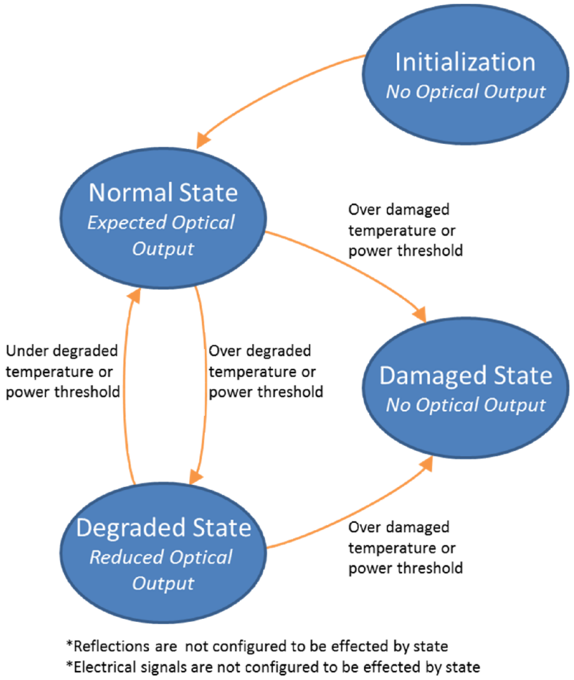
L.9 Appendix C – Component Use Case

L.9.1 Respond to an Optical Packet in the Isolator

Optical packet arrives at the isolator. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the isolator. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated optical output signal based on the input signal, direction of input and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

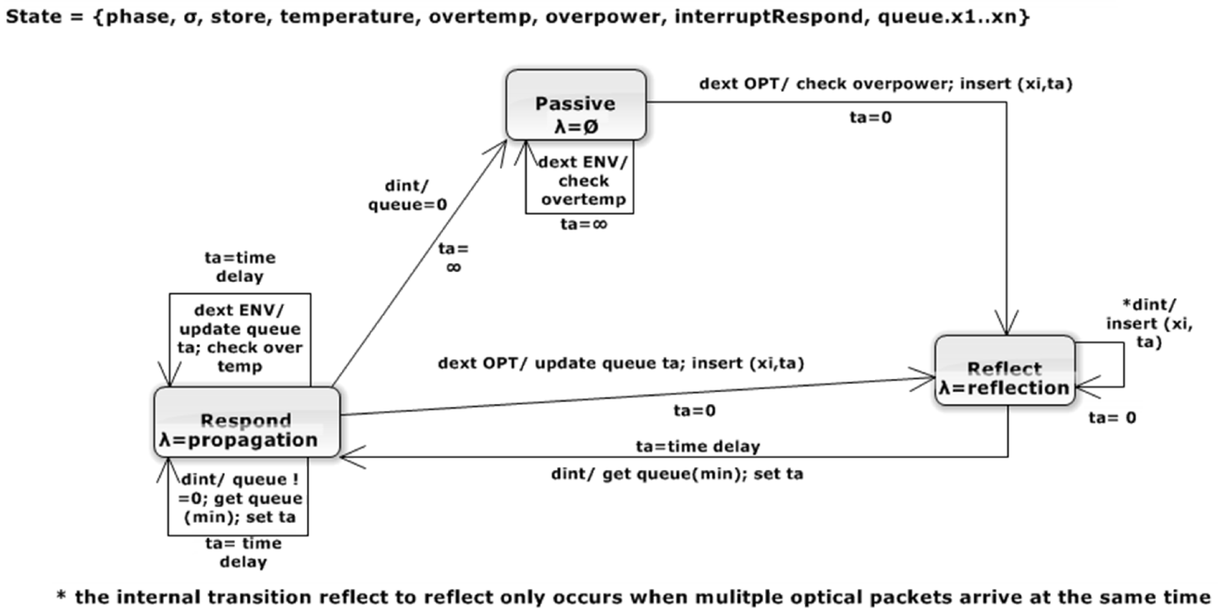
- Identified Alternative Uses Cases
    - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 114. Component states.


State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}

Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
ta=0
ta= 0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue != 0; get queue (min); set ta
ta= time delay

*** the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time**

Figure 115. Isolator phase transition diagram.

L.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly.
- Optical packet entered and removed from queue in proper sequence.
- Overpower condition properly recognized and recorded.
- Optical packet attenuated properly to the limit of accuracy.
- Optical packet propagated out the correct port at the correct time.

L.9.3 Respond to an Environmental Packet in the Isolator

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

L.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

L.10 Isolator Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Isolator Test Cases*

		Inject Ports				Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Env**	**Notes**	opt #	**env #**
Passive	1	1	0	0	single	1	0
	2	0	1	0	single	2	0
	3	0	0	1	single	2	1
	4	1	1	0	same time	4	1
	5	1	1	0	differ time	6	1
	6	1	1	1	same time	8	2
	7	1	1	1	differ time	10	3
	8	0	1	1	same time	11	4
	9	0	1	1	differ time	12	5
	10	1	0	1	same time	13	6
	11	1	0	1	differ time	14	7
	20	2	0	0	same time	16	7
	21	0	2	0	same time	18	7

	22	2	2	0	same time	22	7
	23	2	2	0	differ time	26	7
	24	2	2	1	same time	30	8
	25	2	2	1	differ time	34	9
	26	0	2	1	same time	36	10
	27	0	2	1	differ time	38	11
	28	2	0	1	same time	40	12
	29	2	0	1	differ time	42	13
totals		21	21	13	42		
Respond	41	2	0	0	single	44	13
	42	0	2	0	single	46	13
	43	1	0	1	single	47	14
	44	2	1	0	same time	50	14
	45	2	1	0	differ time	53	14
	46	2	1	1	same time	56	15
	47	2	1	1	differ time	59	16
	48	0	2	1	same time	61	17
	49	0	2	1	differ time	63	18
	50	2	0	1	same time	65	19
	51	2	0	1	differ time	67	20
	60	3	0	0	same time	70	20
	61	0	3	0	same time	73	20
	62	3	2	0	same time	78	20
	63	3	2	0	differ time	83	20
	64	3	2	1	same time	88	21
	65	3	2	1	differ time	93	22
	66	0	3	1	same time	96	23
	67	0	3	1	differ time	99	24
	68	3	0	1	same time	102	25
	69	3	0	1	differ time	105	26
totals		36	27	13	63		
	TC1	1	0	2	single	106	28
	TC2	1	0	2	single	107	30
	TC3	1	0	2	single	108	32
	TC4	1	0	2	single	109	34
	TC5	1	0	2	single	110	36
	TC6	1	0	2	single	111	38
	TC7	1	0	2	single	112	40
totals		7	0	14	21		

L.11 References

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). Optical isolator tutorial. Retrieved September 16, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6178

Appendix M - Laser

M.1 Device Description:

The laser is an optical oscillator that contains an amplification medium and emits coherent light via an output coupler. Spontaneous emission, or injected seed light, is amplified with each pass through the amplification portion of the oscillator. The geometry of oscillator dictates which optical wavelengths and modes are supported. Lasers come in many forms and are generated by a multitude of materials used as the active medium. Different forms include solid-state lasers, gas lasers, dye lasers, x-ray and free-electron lasers. They can also produce pulsed or continuous beams of light and can produce specific polarization of light by using embedded polarizers or Brewster windows.

Active materials include (solid-state): ruby, alexandrite, sapphire, yttrium aluminum garnet, gadolinium gallium garnet, yttrium lithium fluoride, transition-metal and lanthanide-metal ions; (gas): helium-neon, argon, krypton, carbon dioxide, xenon-chloride, hydrogen fluoride; (dyes): polymethine dyes, xanthenes dyes; and exotic materials used in the high energy x-ray and free-electron lasers (Saleh & Teich, 1991). See Figure 1 for examples of lasers.

Figure 116. Example of lasers (ThorLabs, 2013).

The laser is a unidirectional optical component with one optical port and one electrical control port (unidirectional in the sense that it is not designed to pass optical signals through the device). The optical port is the output port for the coherent optical pulses or beam created inside the device. Optical signals arriving at the optical port are reflected back down the optical path after suffering an amount of attenuation. The laser is sensitive to the power of the optical signals that are received by the component. If the optical power of a pulse exceeds a defined threshold, the laser may become permanently damaged which changes its attenuation and output characteristics. Similarly, the laser is sensitive to the temperature in the environment in which it operates, as temperature changes alter the physical dimensions of the device. If the temperature exceeds defined thresholds, the laser may become temporarily degraded or permanently damaged which changes the output wavelength. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the laser is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the laser using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the laser. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined by the SME. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the expected behavior and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

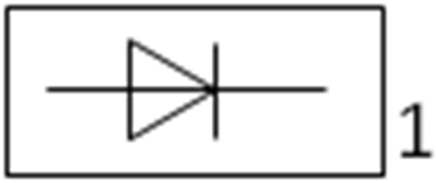


Figure 117. Symbol for the laser in the QKD system architecture.

M.2 Laser Conceptual Model

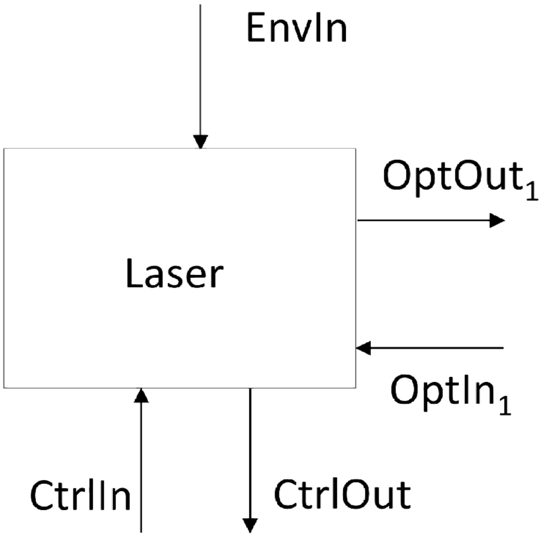


Figure 118. Laser conceptual model.

The conceptual model for a laser consists of one optical input port {OptIn$_1$ }, one optical output port {OptOut$_1$}, one environmental input port {EvnIn} and one electrical controller input port and one electrical controller output port {CtrlIn, CtrlOut}. The environmental port allows external sources to communicate changes in the operational environment to the laser. The electrical controller ports allow for control inputs to the controller and responses from the laser to the higher system functions.

In comparison to the laser symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. The electrical control port is not shown for clarity in Figure 2, and is also decomposed in the model into an input port and an output port. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the laser, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 3 will instantaneously generate a reflected emitting out of OptOut_1.

The laser must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation and output. External environmental messages sent to the device convey the temperature of the operational environmental so the laser can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the attenuation and output changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

The laser is unique among the optical components in that it creates optical messages. The laser receives a command from the controller to ‘fire’ a pulse. This is an abstraction how a laser actually monitors an electrical voltage and emits a pulse when the voltage exceeds a trigger voltage. It does not emit a pulse until the voltage drops below the trigger voltage and then exceeds the trigger voltage.

Upon receipt of a ‘fire’ or ‘laser on’ message, the laser creates an optical pulse message with predefined characteristics and sends it out the optical port. The time between receipt of the message and emission of the pulse has a random factor supplied by a random Gaussian draw. In the case of the demonstration architecture, the optical pulse is modeled on the output of the ID Quantique ID300 laser (ID Quantique SA, 2011).

M.3 Mathematical Model

There is no detailed mathematical description of the laser in Section 11.8.

M.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the laser.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Calculate the reflected power of the signal and send its output with the same port number.

When an environmental message arrives:

- Update the CurrrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

When a control message arrives:

- Determine if it is an "on" or "off" message.
- Output optical packets if the message is an "on" or stop packet output if it is an "off" message
- Respond to the controller with an acknowledgement message.

M.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the laser in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the laser at the same time.

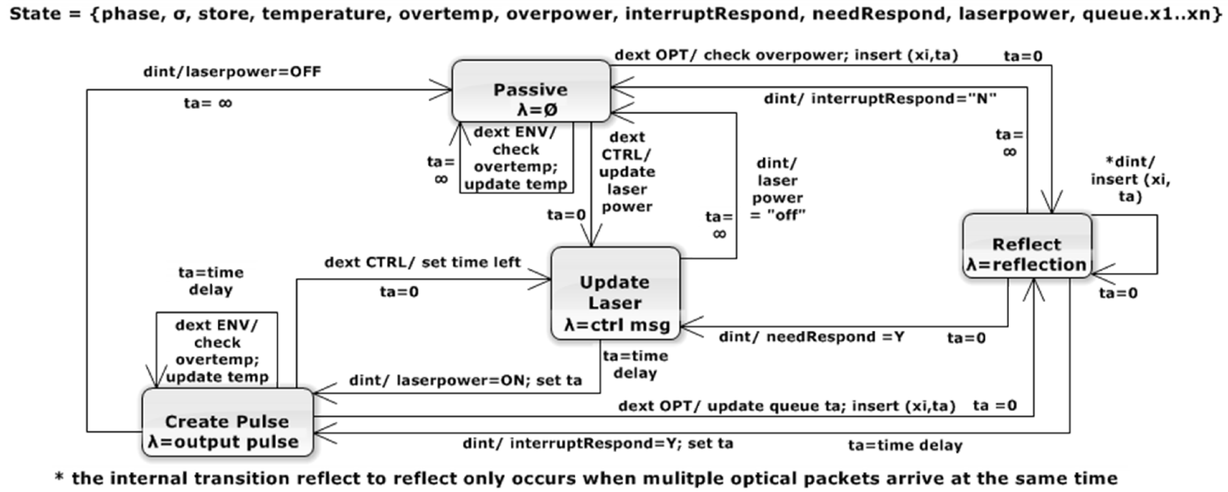

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, laserpower, queue.x1..xn}
Passive
λ=Ø
dint/laserpower=OFF
ta= ∞
dext OPT/ check overpower; insert (xi,ta)
ta=0
dint/ interruptRespond="N"
ta= ∞
dext ENV/ check overtemp; update temp
ta= ∞
dext CTRL/ update laser power
ta=0
dint/ laser power = "off"
ta= ∞
*dint/ insert (xi, ta)
Reflect
λ=reflection
ta=0
ta=time delay
dext CTRL/ set time left
ta=0
Update Laser
λ=ctrl msg
dint/ needRespond =Y
ta=0
dext ENV/ check overtemp; update temp
ta=time delay
dint/ laserpower=ON; set ta
Create Pulse
λ=output pulse
dext OPT/ update queue ta; insert (xi,ta)
ta =0
dint/ interruptRespond=Y; set ta
ta=time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 119. Laser phase transition diagram.

M.6 Event-Trace Diagram

This section shows various examples of packets entering the laser. The tables list the states the laser proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the laser, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Interrupt Respond: shows the value of the interrupt respond variable
- Need Respond: shows the value of the need respond variable
- Laser Power: shows the value of the laser power variable
- Queue: contents of the queue for that state

- Notes: any notes for that state

M.6.1 CASE I: Initial Passive with Single Control Packet Arriving at Time 0

Table 48. *Case I state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	need respond	laser power	queue (x*i*, *tp*)	Notes:
	1 ctrl	0 env	0 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	
0	s0	exit	passive	0	ctrl	c	n	n	n	n	off	null	
0	s1	entry	update laser	0	ctrl	c	n	n	n	n	off	null	
0	s1	exit	update laser	2x10^8	ctrl	c	n	n	n	n	on	null	
0	s2	entry	create pulse	2x10^8	ctrl	c	n	n	n	n	on	null	
2x10^8	s2	exit	create pulse	0	ctrl	c	n	n	n	n	off	null	
2x10^8	s3	entry	passive	inf	ctrl	c	n	n	n	n	off	null	

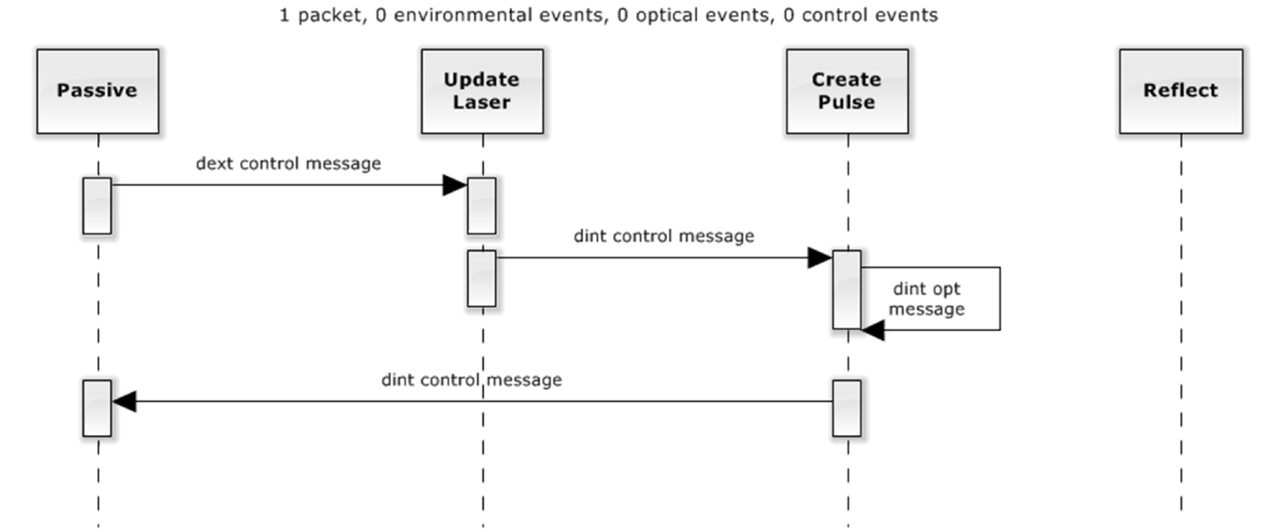


Figure 120. Case I sequence diagram.

M.6.2 CASE II: Initial Passive with Single Control Packet Arriving at Time 0 and 1 Optical Packet Arriving Time 1x10^8

Table 49. *Case II state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	need respond	laser power	queue (x*i*, *tp*)	Notes:
	1 ctrl	0 env	0 opt	1 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	

0	s0	exit	passive	0	ctrl	c	n	n	n	n	off	null	
0	s1	entry	update laser	0	ctrl	c	n	n	n	n	off	null	
0	s1	exit	update laser	2x10^8	ctrl	c	n	n	n	n	on	null	
0	s2	entry	create pulse	2x10^8	ctrl	c	n	n	n	n	on	null	
1x10^8	s2	exit	create pulse	0	ctrl	c	n	n	y	n	on	null	dext at e= 1x10^8, 1 ctrl packet "status"
1x10^8	s3	entry	update laser	0	ctrl	c	n	n	y	n	on	null	
1x10^8	s3	exit	update laser	1x10^8	ctrl	c	n	n	y	n	on	null	
1x10^8	s4	entry	create pulse	1x10^8	ctrl	c	n	n	y	n	on	null	
2x10^8	s4	exit	create pulse	inf	ctrl	c	n	n	n	n	off	null	
2x10^8	s5	entry	passive	inf	ctrl	c	n	n	n	n	off	null	

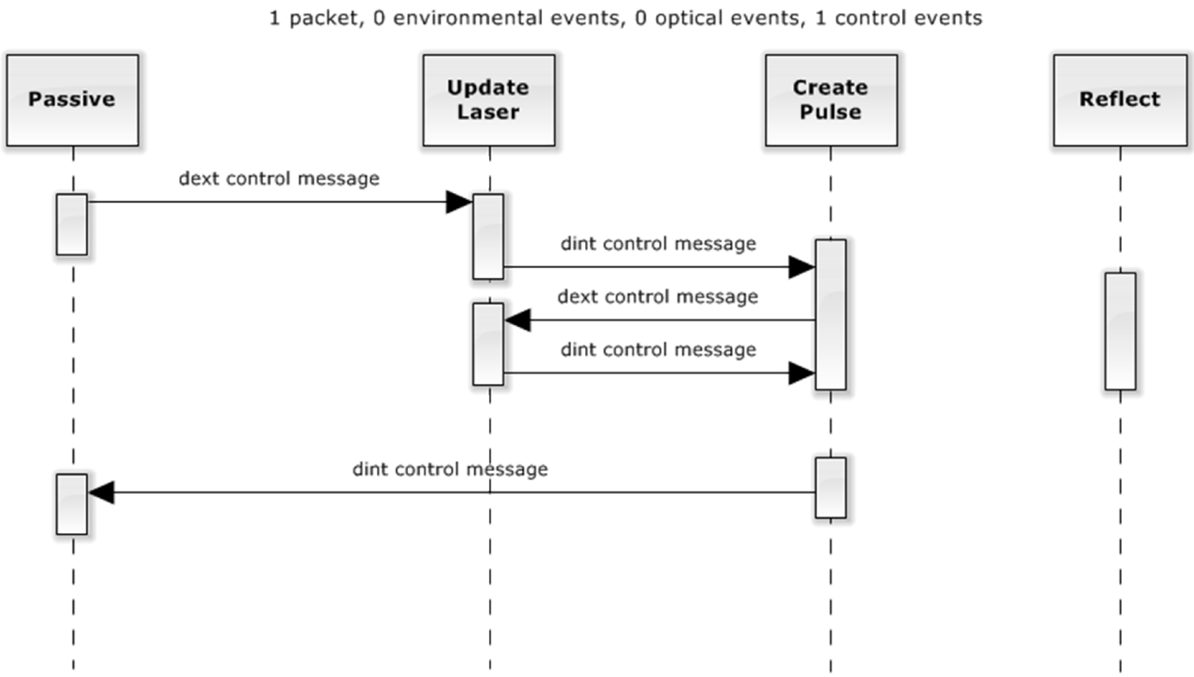


Figure 121. Case II sequence diagram.

M.6.3 CASE III: Initial Passive with Single Control Packet Arriving at Time 0 and 1 Environmental Packet Arriving at Time 1x10^8

Table 50. *Case III state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	need respond	laser power	queue (x*i*, *tp*)	Notes:
	1	1 env	0 opt	0 ctrl									

	ctrl												
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	
0	s0	exit	passive	0	ctrl	c	n	n	n	n	off	null	
0	s1	entry	update laser	0	ctrl	c	n	n	n	n	off	null	
0	s1	exit	update laser	2x10^8	ctrl	c	n	n	n	n	on	null	
0	s2	entry	create pulse	2x10^8	ctrl	c	n	n	n	n	on	null	
1x10^8	s2	exit	create pulse	1x10^8	env	c	n	n	y	n	on	null	dext at e= 1x10^8, 1 env packet overtemp
1x10^8	s3	entry	create pulse	1x10^8	env	c	n	n	y	n	on	null	
2x10^8	s3	exit	create pulse	inf	env	c	n	n	n	n	off	null	
2x10^8	s4	entry	passive	inf	env	c	n	n	n	n	off	null	

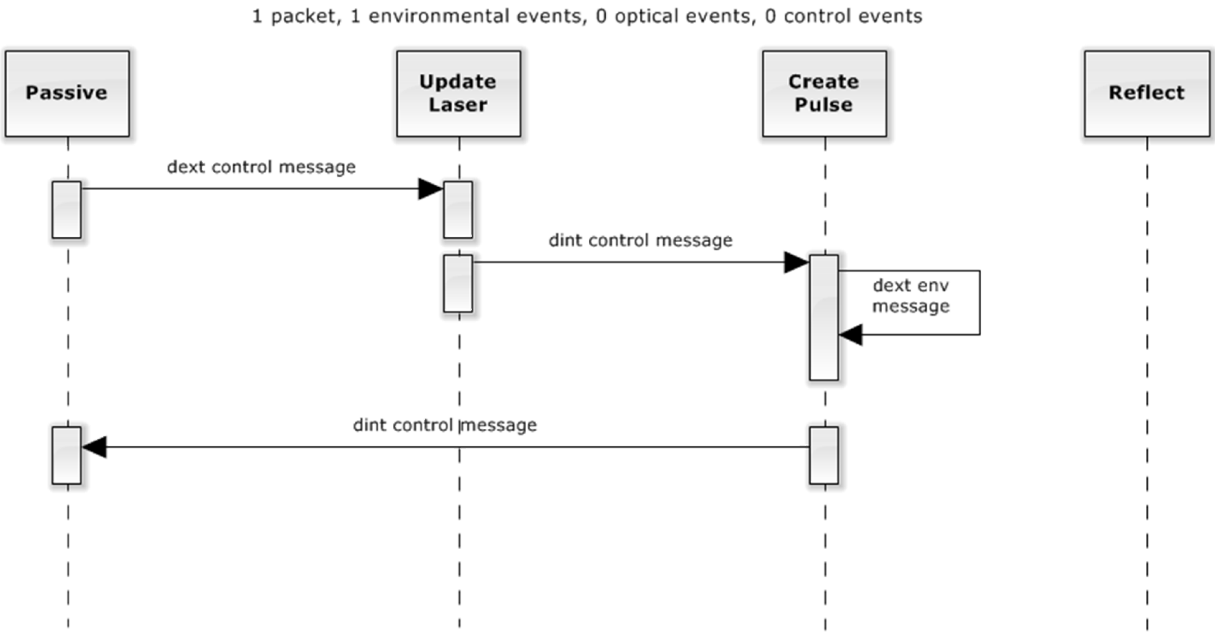


Figure 122. Case III sequence diagram.

M.6.4 CASE IV: Initial Passive with Single Control Packet Arriving at Time 0 and 1 Control Packet Arriving at Time 1x10^8

Table 51. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	need respond	laser power	queue (*xi*, *tp*)	Notes:
	1 ctrl	0 env	1 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	

0	s0	exit	passive	0	ctrl	c	n	n	n	n	off	null	
0	s1	entry	update laser	0	ctrl	c	n	n	n	n	off	null	
0	s1	exit	update laser	2x10^8	ctrl	c	n	n	n	n	on	null	
0	s2	entry	create pulse	2x10^8	ctrl	c	n	n	n	n	on	null	
1x10^8	s2	exit	create pulse	0	x1	c	n	n	y	n	on	null	dext at e= 1x10^8, 1 opt packet
1x10^8	s3	entry	reflect	0	x1	c	n	n	y	n	on	null	
1x10^8	s3	exit	reflect	1x10^8	x1	c	n	n	y	n	on	null	
1x10^8	s4	entry	create pulse	1x10^8	x1	c	n	n	y	n	on	null	
2x10^8	s4	exit	create pulse	inf	x1	c	n	n	n	n	off	null	
2x10^8	s5	entry	passive	inf	x1	c	n	n	n	n	off	null	

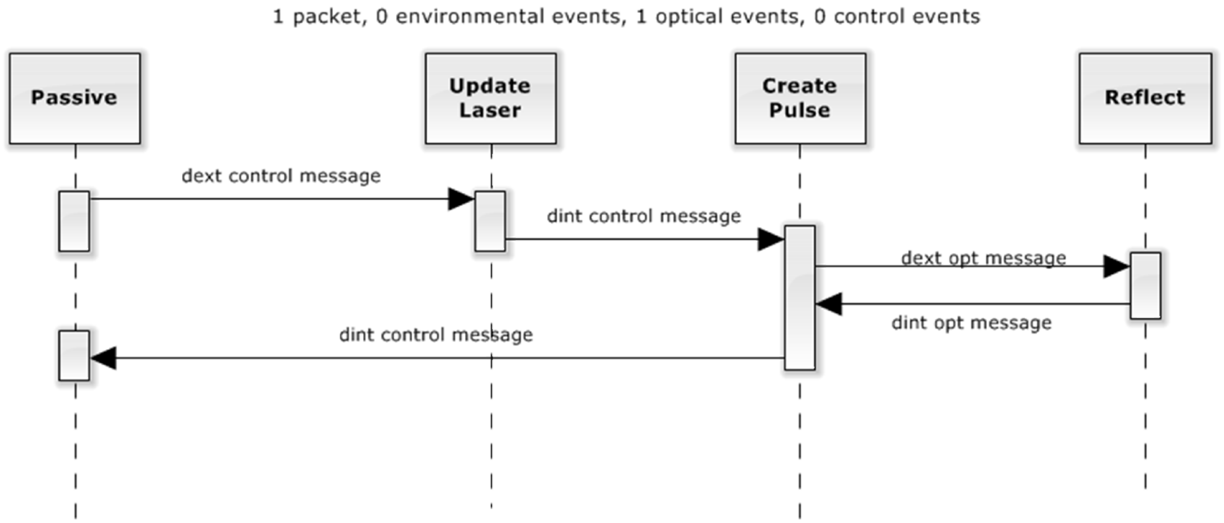


Figure 123. Case IV sequence diagram.

M.7 Laser Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.

- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, “needRespond”,“laserpower”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“laserpower” = flag variable set when device is turned on
Peak power = full width, half maximum power calculation of the pulse

For the laser we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS$_{laser}$* = (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator

phase = control state that keeps track of the internal phase of the attenuator
phase = {"passive", "reflect", "respond", "update detector"}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
needRespond= flag variable set when both Reflect and UpdateDetector respond to inputs
laserpower = flag variable set when device is turned on
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current x input value(s) (p,v)
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event i
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
ctrlOutput = variable that stores the output control message response
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
autoPulseInterval= automatic pulse interval in seconds
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
queue_clear() = method that clears the queue of all entries
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by e
ctrlMsg() = method that generates a response message to received control messages
outputMsg() = method that generates the response message to received optical packets
insert_event_q() = method that inserts the current (x_i, time delay_i) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay_i) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse

calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“OptIn_1”, “EnvIn”, “CtrlIn”} with
X_M = {(“OptIn_1”, V_{opt}), (“EnvIn”, V_{env}), (“CtrlIn”, V_{ctrl})} is the set of input ports and values.

OutPorts = {“OptOut_1”, “CtrlOut”} with
Y_M = {(“OptOut_1”, $Y_{OptOut1}$), (“CtrlOut”, $Y_{CtrlOut}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond, needRespond, laserpower, queue*} = {{“passive”, “reflect”, “create pulse”, “update laser”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x{“Y”,”N”} x {“on”, “off”} x *V*}

External Transition Function:

δ_{ext}(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, laserpower ,needRespond, queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =
(“update laser”, 0, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, needRespond, laserpower,queue.x*1*..xn*)

```
if phase = “passive” and p = “CtrlIn”
  if messagebag.status= “statuscheck “
    laserpower = “off”
  if messagebag.status = “laserfire”
    laserpower = “on”
  ctrlOutput = ctrlMsg(store)
```

```
(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                                   laserpower, queue.x1..xn)
  if phase = “passive” and p = “OptIn$_1$”
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower =  “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                                   laserpower, queue.x1..xn)
 if phase = “create pulse” and p = “OptIn$_1$”
    update_delay(queue)
    for messagebag != null
     current = messagebag_first()
     if current.value.power > damaged.power
       overpower =  “Y”
     insert_event_q(current)
     remove_event_m(current)
   queue.current = queue_need_reflected()
   reflect = (queue.current.p), calcReflected(queue.current.v))
   mark_reflected(queue.current)
   interruptRespond= “Y”
   timeLeftRespond = timeLeftRespond - e

 (“update laser”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                                   laserpower, queue.x1..xn)
  if phase = “create pulse” and p = “CtrlIn”
   if  messagebag.status= “statuscheck “
    ctrlOutput = ctrlMsg(store)
    interruptRespond= “Y”
    timeLeftRespond = timeLeftRespond – e
   if messagebag.status = “laserfire”
    interruptRespond= “Y”
    timeLeftRespond = timeLeftRespond – e

 (“create pulse”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                      needRespond, laserpower, queue.x1..xn)
   if phase = “create pulse” and p = “EnvIn”
    timeLeftRespond = time_delay- e
```

```
      temperature = messagebag.temperature
      if temperature > damage.temp
        overtemp = "Y"
      time.delay = timeLeftRespond

("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                              laserpower, queue.x1..xn)
    if phase = "passive" and p = "EnvIn"
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"

 (phase, $\sigma - e$, store, temperature, overtemp, overpower, laserpower)
          otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interrupRespond, needRespond,
                                                                          laserpower) =

  ("reflect", 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                              laserpower, queue.x1..xn)
    if phase = "reflect" and need.reflect != null
      need.reflect = queue_need_reflected()
      current = need.reflect
      reflect = (current.p), calcReflected(current.v))
      mark_reflected(current)

  ("create pulse", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                               needRespond, laserpower, queue.x1..xn)
    if phase = "reflect" and interruptRespond = "Y" and needreflect = 0
       need.reflect = queue_need_reflected()
       if interruptRespond = "Y"
       time_delay = timeLeftRespond
      queue_clear()

  ("update laser", 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                              laserpower, queue.x1..xn)
    if phase = "reflect" and needRespond = "Y"
     if  messagebag.status= "statuscheck "
       laserpower = "off"
       ctrlOutput = ctrlMsg(store)
     if messagebag.status = "laserfire"
       laserpower = "on"
     queue_clear()
```

```
("passive", ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                        laserpower, queue.x1..xn)
  if phase = "reflect" and interruptRespond = "N"
    queue_clear()

("passive", ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                        laserpower, queue.x1..xn)
  if phase = "create pulse"
    needRespond = "N"
    interruptRespond= "N"
    laserpower = "OFF"


("create pulse", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                            needRespond, laserpower, queue.x1..xn)
  if phase = "update laser" and (laserpower = "on" or interruptRespond= "Y")
    if interruptRespond = "Y"
    time.delay = timeLeftRespond
   else
     time.delay = autoPulseInterval


("passive", ∞, store, temperature, overtemp, overpower, overpower, interruptRespond,
                                            needRespond, laserpower, queue.x1..xn)

  if phase = "update laser" and laserpower = "off"
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

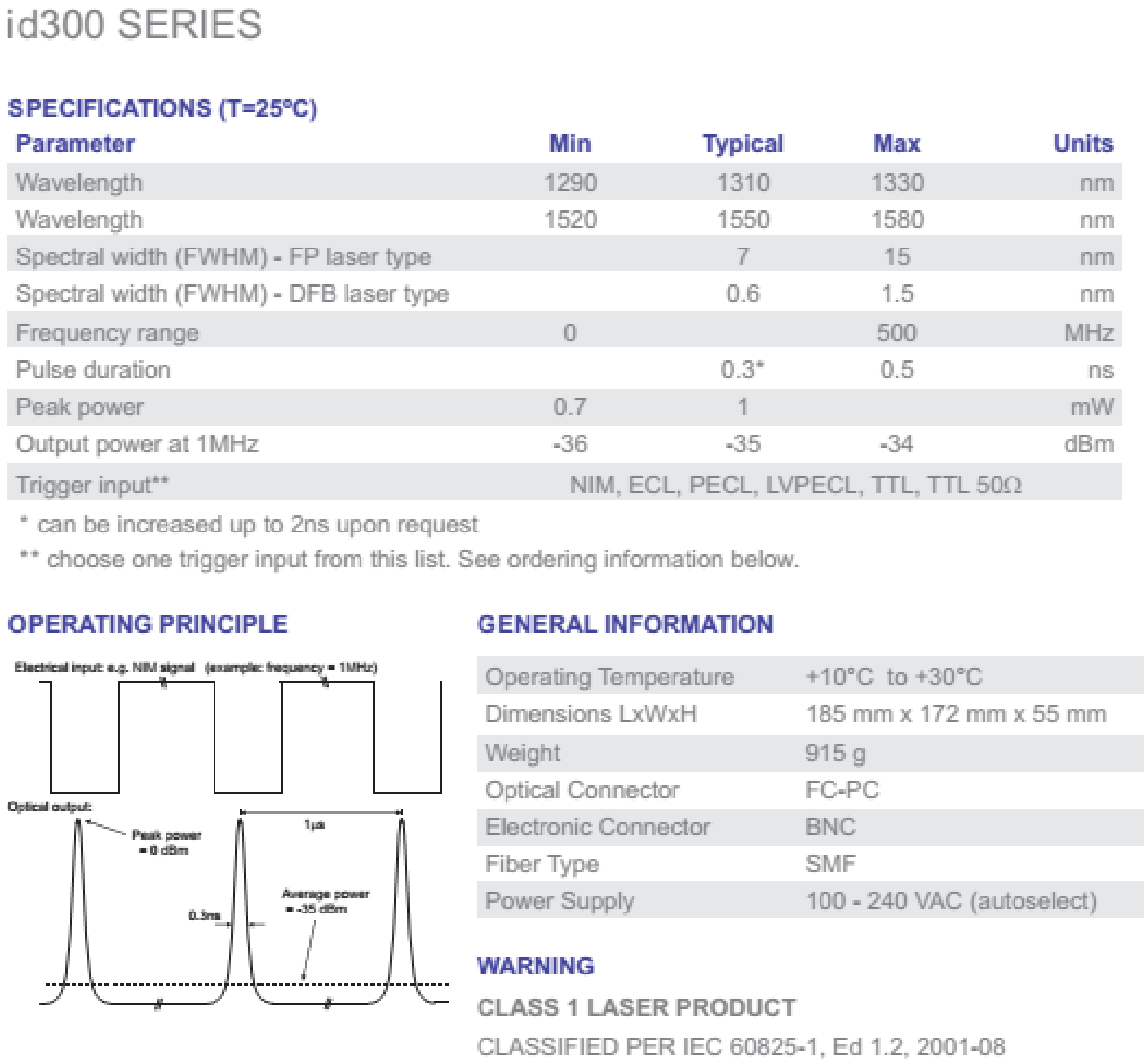
Output Function:

```
λ(phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                        laserpower, queue) =
  (reflect.p, reflect.v)
    if phase = "reflect"

  (Output1, outputPulse)
    if phase = "create pulse"

  ("CtrlOut", ctrlOutput)
   if phase = "update laser"

  ∅ (null output)
   otherwise;
```

Time advance Function:

ta(*phase*, *σ*, *store*, *temperature*, *overtemp*, *overpower*, *interruptRespond*, *needRespond*, *laserpower*, *queue*) = *σ*;

M.8 Mathematical Model



id300 SERIES

SPECIFICATIONS (T=25°C)

Parameter	Min	Typical	Max	Units
Wavelength	1290	1310	1330	nm
Wavelength	1520	1550	1580	nm
Spectral width (FWHM) - FP laser type		7	15	nm
Spectral width (FWHM) - DFB laser type		0.6	1.5	nm
Frequency range	0		500	MHz
Pulse duration		0.3*	0.5	ns
Peak power	0.7	1		mW
Output power at 1MHz	-36	-35	-34	dBm
Trigger input**	NIM, ECL, PECL, LVPECL, TTL, TTL 50Ω			

* can be increased up to 2ns upon request

** choose one trigger input from this list. See ordering information below.

OPERATING PRINCIPLE

GENERAL INFORMATION

Operating Temperature	+10°C to +30°C
Dimensions LxWxH	185 mm x 172 mm x 55 mm
Weight	915 g
Optical Connector	FC-PC
Electronic Connector	BNC
Fiber Type	SMF
Power Supply	100 - 240 VAC (autoselect)

WARNING

CLASS 1 LASER PRODUCT

CLASSIFIED PER IEC 60825-1, Ed 1.2, 2001-08

(ID Quantique SA, 2011)

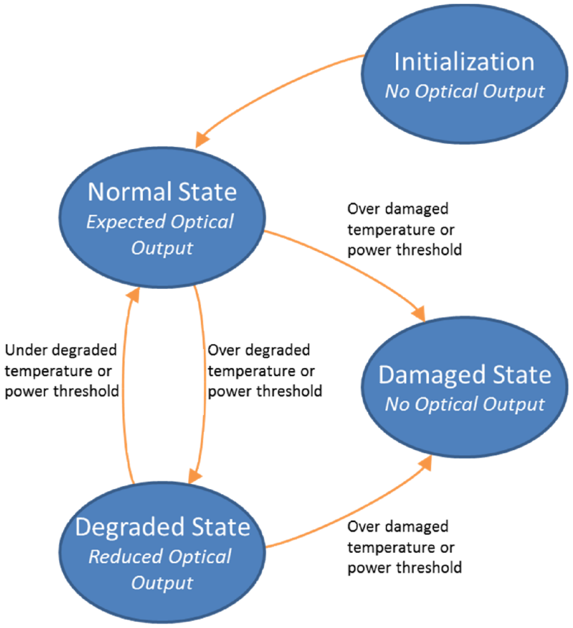
M.9 Component Use Case

M.9.1 Respond to an Optical Packet in the Laser

Optical packet arrives at laser. A portion of optical packet reflects back down incoming optical line. Check to see if optical packet overpowers the EVOA. Records overpower condition, if applicable.

- Identified Alternative Uses Cases
  - Respond to a control message
  - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 124. Component states.

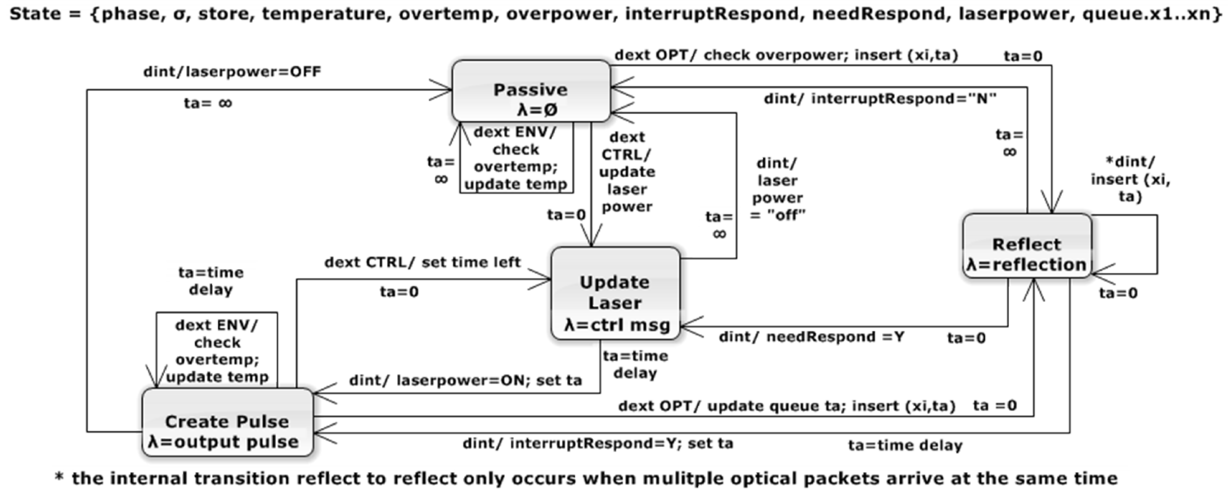

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, laserpower, queue.x1..xn}
dint/laserpower=OFF
ta= ∞
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dint/ interruptRespond="N"
ta= ∞
dext ENV/ check overtemp; update temp
ta= ∞
dext CTRL/ update laser power
ta=0
dint/ laser power = "off"
ta= ∞
*dint/ insert (xi, ta)
Reflect
λ=reflection
ta=0
ta=time delay
dext CTRL/ set time left
ta=0
Update Laser
λ=ctrl msg
dint/ needRespond =Y
ta=0
dext ENV/ check overtemp; update temp
ta=time delay
dint/ laserpower=ON; set ta
dext OPT/ update queue ta; insert (xi,ta)
ta =0
Create Pulse
λ=output pulse
dint/ interruptRespond=Y; set ta
ta=time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 125. Laser phase transition diagram.

M.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Overpower condition properly recognized and recorded

M.9.3 Respond to an Environmental Packet in the Laser

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
  - None

M.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

M.9.5 Respond to a Control Message in the Laser

Control Message arrives at the component. Component decodes message properly. Records change in condition or state, if applicable. Change component function if in degraded or damaged state or by change in component condition, if necessary.

- Assumptions
    - Component has completed initialization sequence at least once

M.9.6 Respond to Control Message End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

M.10 Laser Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Laser Test Cases*

		Inject Ports				Running Totals		
Phase	**Case**	**Opt1**	**Ctrl**	**Env**	**Notes**	opt #	**env #**	**ctrl #**
Passive	1	1	0	0	single	1	0	0
	2	0	1	0	single	1	0	1
	3	0	0	1	single	1	1	1
	4	1	1	0	same time	2	1	2
	5	1	1	0	differ time	3	1	3
	6	1	1	1	same time	4	2	4
	7	1	1	1	differ time	5	3	5
	8	0	1	1	same time	5	4	6
	9	0	1	1	differ time	5	5	7
	10	1	0	1	same time	6	6	7
	11	1	0	1	differ time	7	7	7
	20	2	0	0	same time	9	7	7
	21	0	1	0	same time	9	7	8
	22	2	1	0	same time	11	7	9
	23	2	1	0	differ time	13	7	10
	24	2	1	1	same time	15	8	11
	25	2	1	1	differ time	17	9	12
	26	0	1	1	same time	17	10	13
	27	0	1	1	differ time	17	11	14
	28	2	0	1	same time	19	12	14
	29	2	0	1	differ time	21	13	14
totals		21	14	13	35			
Respond	41	2	0	0	single	23	13	14
	42	1	1	0	single	24	13	15
	43	1	0	1	single	25	14	15
	44	2	1	0	same time	27	14	16
	45	2	1	0	differ time	29	14	17

46	2	1	1	same time	31	15	18
47	2	1	1	differ time	33	16	19
48	1	1	1	same time	34	17	20
49	1	2	1	differ time	35	18	22
50	2	0	1	same time	37	19	22
51	2	0	1	differ time	39	20	22
60	3	0	0	same time	42	20	22
61	1	1	0	same time	43	20	23
62	3	1	0	same time	46	20	24
63	3	1	0	differ time	49	20	25
64	3	1	1	same time	52	21	26
65	3	1	1	differ time	55	22	27
66	1	1	1	same time	56	23	28
67	1	1	1	differ time	57	24	29
68	3	0	1	same time	60	25	29
69	3	0	1	differ time	63	26	29
totals	42	15	13	57			
TC1	1	1	2	single	64	28	30
TC2	1	1	2	single	65	30	31
TC3	1	1	2	single	66	32	32
TC4	1	1	2	single	67	34	33
TC5	1	1	2	single	68	36	34
TC6	1	1	2	single	69	38	35
TC7	1	0	2	single	70	40	35
totals	7	6	14	13			

Notes: 23 - INIT control message sent; OPT1 & Ctrl - differ time - Passive
24 - INIT control message sent - OPT1 & Ctrl - same time - Passive
63 - INIT control message sent - OPT1 & Ctrl - same time - Respond
66 - INIT control message sent - Ctrl & ENV - same time - Respond

M.11 References

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). Coherent sources. Retrieved October 10, 2013, from http://www.thorlabs.com/navigation.cfm?guide_id=31

Appendix N - Panda Polarization Maintaining Optical Fiber (PM Fiber)

N.1 Device Description:

PM fiber is used in optical components where it is required that optical polarization is maintained. Like single-mode fiber, it is a cylindrical optical waveguide made from a low-loss material, such as silica glass. Light that is properly introduced into the fiber maintains its linear polarization during propagation. In the case of Corning fiber, using two stressors within in the fiber creates high birefringence resulting in the fiber maintaining the polarization (Corning, 2013). See Figure 1 for a typical cross-sectional view of a PANDA fiber.

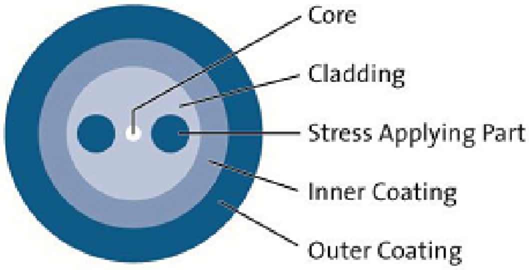


Figure 126. Cross-section of a PANDA PM fiber (Corning, 2013).

PM fiber is a specialty fiber that intentionally uses the strong birefringence in two modes. Light travels down one of the modes faster than down the other (fast and slow axes). If the input light is polarized and oriented along either mode, it maintains its polarization state even if the fiber is stressed (OZOptics, 2014). Typically, PM fiber is used in components that cannot have drift in the polarization state (such as fiber interferometers and some fiber lasers) (RPPhotonics, 2013) .

The PM fiber is a bidirectional optical component with two optical ports. Light entering the primary port is propagated through the fiber, suffering both a slight attenuation from the material of the device and a small propagation delay dependent on the temperature of the fiber. The material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the PM fiber may become permanently damaged which changes its propagation characteristics. Similarly, it is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the PM fiber may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a "normal" operating temperature.

The first step involved with the modeling the PM fiber is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the PM fiber. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the PM fiber to the researcher.

The next step of the modeling effort was to develop a conceptual model of the PM fiber using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the PM fiber. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica

worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

Figure 127. Symbol for Polarization Maintaining Fiber in the QKD system architecture.

N.2 Polarization Maintaining Fiber Conceptual Model

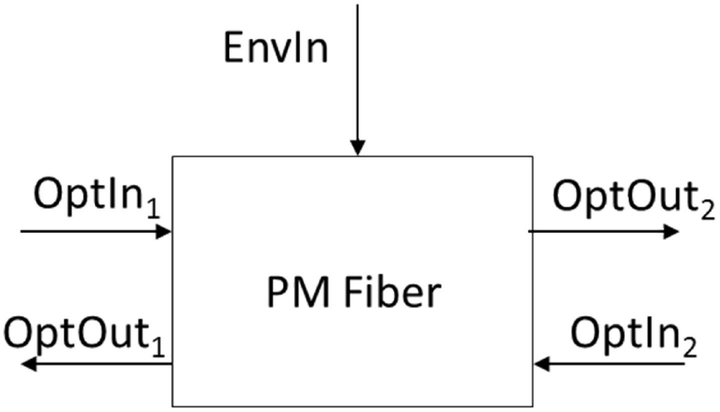


Figure 128. PM fiber conceptual model.

The conceptual model for a PM fiber consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the PM fiber. In comparison to the PM fiber symbol used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the PM fiber, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The PM fiber must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the PM fiber can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the attenuation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

It is important to note that the PM fiber only preserves polarization of light that is injected along the two axes and misalignment between connectors causes unusual effects on the light. The current conceptual model does not take these effects into account and presumes that all light injected into the fiber is aligned properly with the slow and fast axis.

N.3 Mathematical Model

For a detailed mathematical description of the PM fiber, refer to Section 12.8 which contains the Mathematica worksheet provided by the optical physics SME.

N.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the PM fiber.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.

- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.

- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power is less than the minimum power, drop the pulse. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue.
- Remove the packet from the queue; calculate the attenuated output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment and calculate the delay through the fiber based on its length.
- Send the attenuated and delayed output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.

- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

N.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the PM fiber in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the PM fiber at the same time.

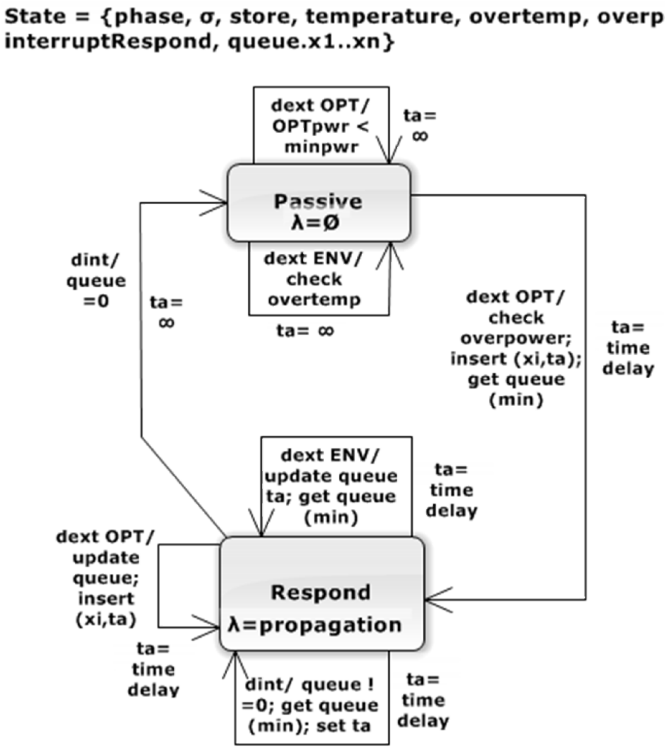

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
dext OPT/ OPTpwr < minpwr
ta= ∞
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dint/ queue =0
ta= ∞
dext OPT/ check overpower; insert (xi,ta); get queue (min)
ta= time delay
dext ENV/ update queue ta; get queue (min)
ta= time delay
dext OPT/ update queue; insert (xi,ta)
ta= time delay
Respond
λ=propagation
dint/ queue != 0; get queue (min); set ta
ta= time delay

Figure 129. PM fiber phase transition diagram.

N.6 Event-Trace Diagram

This section shows various examples of packets entering the PM fiber. The tables list the states the PM fiber proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the PM fiber, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

N.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 52. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (x*i*, *tp*)**	**Notes: assume tp=5**
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s1	entry	respond	5	x1	c	n	n	n	null	
5	s1	exit	respond	inf	x1	c	n	n	n	null	
5	S2	entry	passive	inf	x1	c	n	n	n	null	

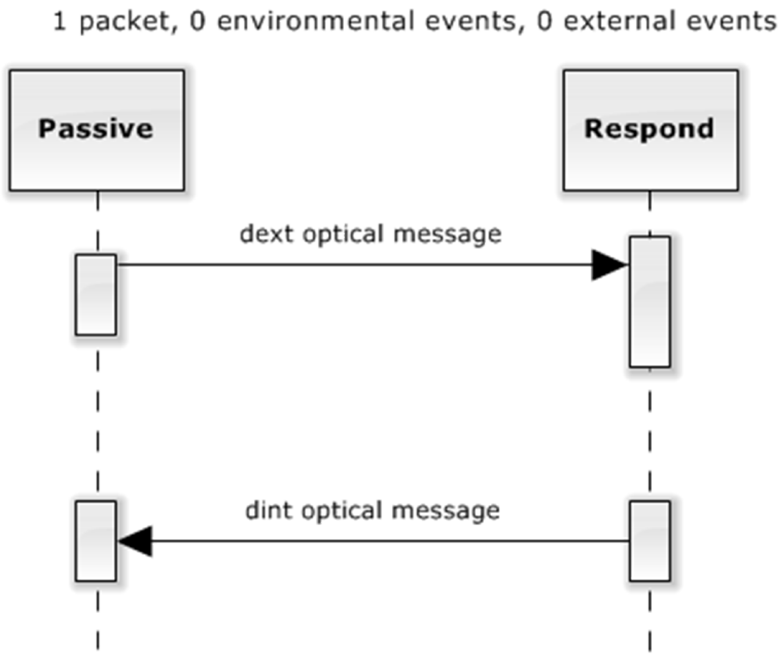


Figure 130. Case I sequence diagram.

N.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 53. *Case II state list.*

	state	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s1	entry	respond	5	x1	c	n	n	n	null	
2	s1	exit	respond	3	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s2	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s2	exit	respond	2	x2	c	n	n	n	null	
5	s3	entry	respond	2	x2	c	n	n	n	null	
7	s3	exit	respond	inf	x2	c	n	n	n	null	
7	s4	entry	passive	inf	x2	c	n	n	n	null	

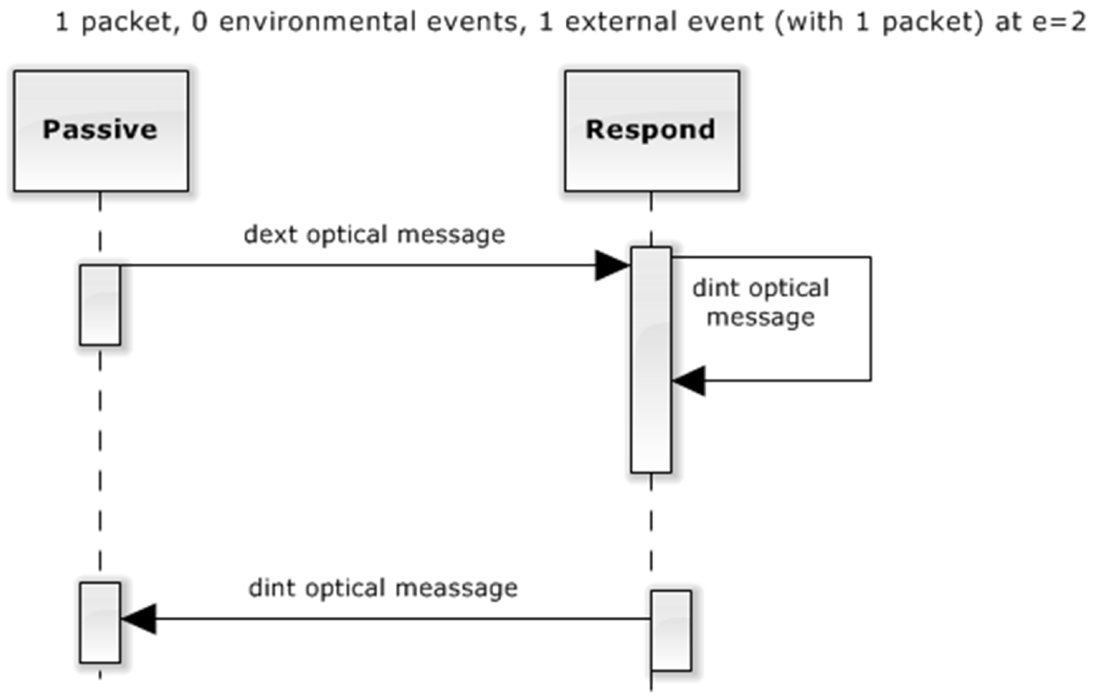


Figure 131. Case II sequence diagram.

N.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 54. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s1	entry	respond	5	x1	c	n	n	n	null	
2	s1	exit	respond	3	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s2	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s2	exit	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	dext at e= 1, 2 optical packets (x3,x4)

3	s3	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s3	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s4	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s4	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s5	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s5	exit	respond	0	x4	c	n	n	n	null	
8	s6	entry	respond	0	x4	c	n	n	n	null	
8	s6	exit	respond	inf	x4	c	n	n	n	null	
8	s7	entry	passive	inf	x4	c	n	n	n	null	

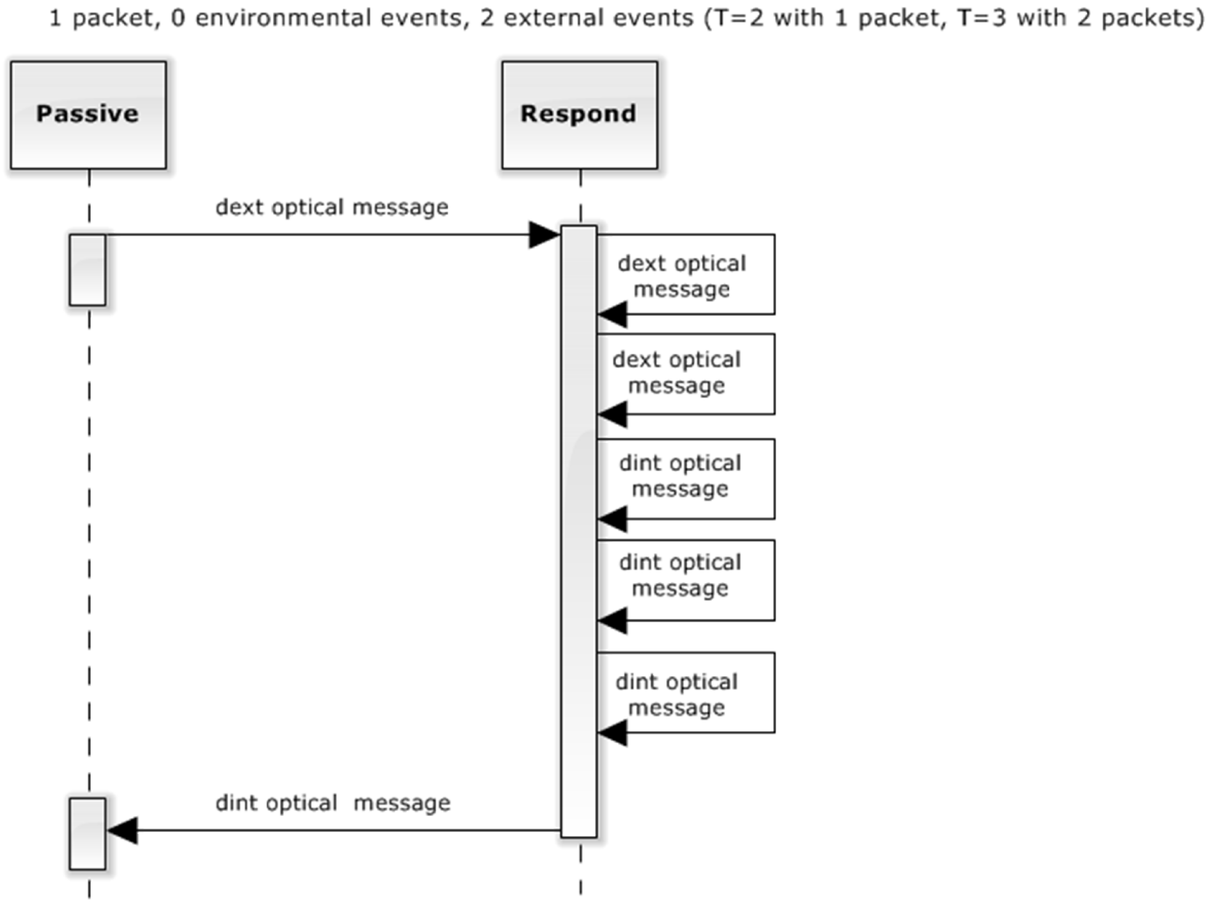


Figure 132. Case III sequence diagram.

N.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 55. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	y	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

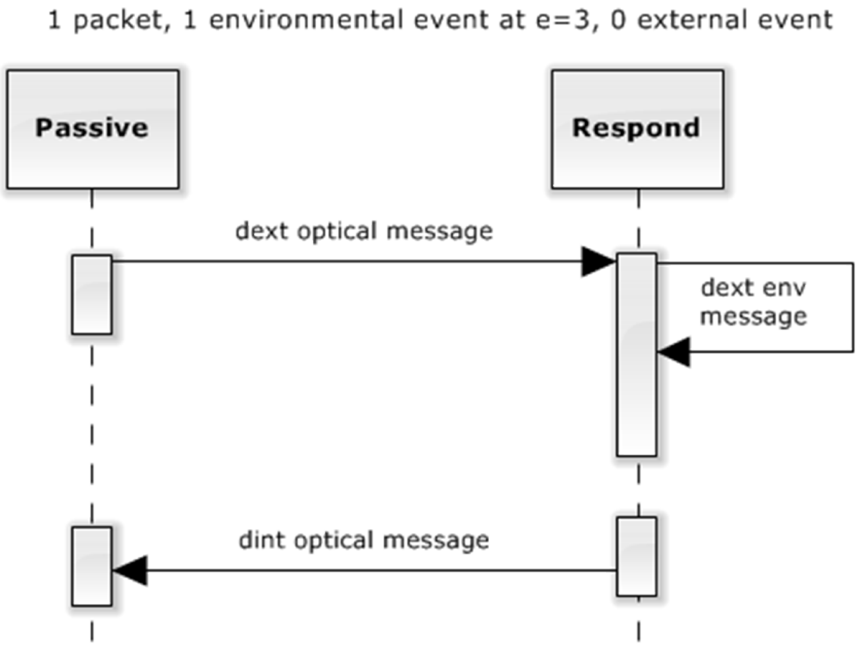


Figure 133. Case IV sequence diagram.

N.7 Single Mod Fiber Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
Peak power = full width, half maximum power calculation of the pulse

For the fixed PM fiber we define:

Parallel-DEVS *atomic M*= $(X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;

$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;

S = set of sequential states;

$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;

$\delta_{int} = S \rightarrow S$ is the internal state transition function;

$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;

$\lambda = S \rightarrow Y^b$ is the output function;

$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;

$Q := \{(s,e) \mid s \in S, 0\leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{PM\,fiber}$* **= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time delay$_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay$_i$) from *messagebag*

calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse

calcAttenDelay() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))

calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))

calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcPolar() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReflected() = method that calculates reflection power of an optical packet

MIN_POWER = global constant that is the minimum acceptable power of an optical packet

q.v = pointer to a value in the queue

q.v_{min} = minimum value in the queue

v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“OptIn$_1$”, “OptIn$_2$”, “EnvIn”} with
X_M = {(“OptIn$_1$”, V_{opt}), (“OptIn$_2$”, V_{opt}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“OptOut$_1$”, “OptOut$_2$”} with
Y_M = {(“OptOut$_1$”, $Y_{OptOut1}$), (“OptOut$_2$”, $Y_{OptOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower*} = {{“passive”, “reflect”, “respond”, “update temperature”, propagate”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”}};

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue, e, (($p_i$,$v_i$),…. ($p_n$,$v_n$))) =
 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = “passive” and p ∈ {“OptIn1”, “OptIn2”}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
```

```
          overpower =  "Y"
        if calcAtten(current.v) > MIN_POWER
         insert_event_q(current)
         remove_event_m(current)
      current = queue_min()
      time.delay = current.time.delay
       if InPort = "OptIn$_1$"
        outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "OptOut$_2$"
       if InPort = "OptIn$_2$"
        outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "OptOut$_1$"
     interruptRespond = "N"

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                      queue.x1..xn)
   if phase = "respond" and p $\in$ {"OptIn$_1$", "OptIn$_2$"}
     update_delay(queue)
     for messagebag != null
       current = messagebag_first()
       if current.value.power > damaged.power
          overpower =  "Y"
       if calcAtten(current.v) > MIN_POWER
        insert_event_q(current)
        remove_event_m(current)
     interruptRespond= "Y"
     timeLeftRespond = timeLeftRespond - e

("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = "passive" and p = "EnvIn"
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"

("respond", time.delay,   store, temperature, overtemp, overpower, interruptRespond,
                                                                      queue.x1..xn)
   if phase = "respond" and p = "EnvIn"
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"
     time.delay = timeLeftRespond


(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
```

```
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if InPort = “$\text{OptIn}_1$”
        outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “$\text{OptOut}_2$”
      if InPort = “$\text{OptIn}_2$”
        outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = “$\text{OptOut}_1$”
      interruptRespond= “N”

 (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = “respond” and size  = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:

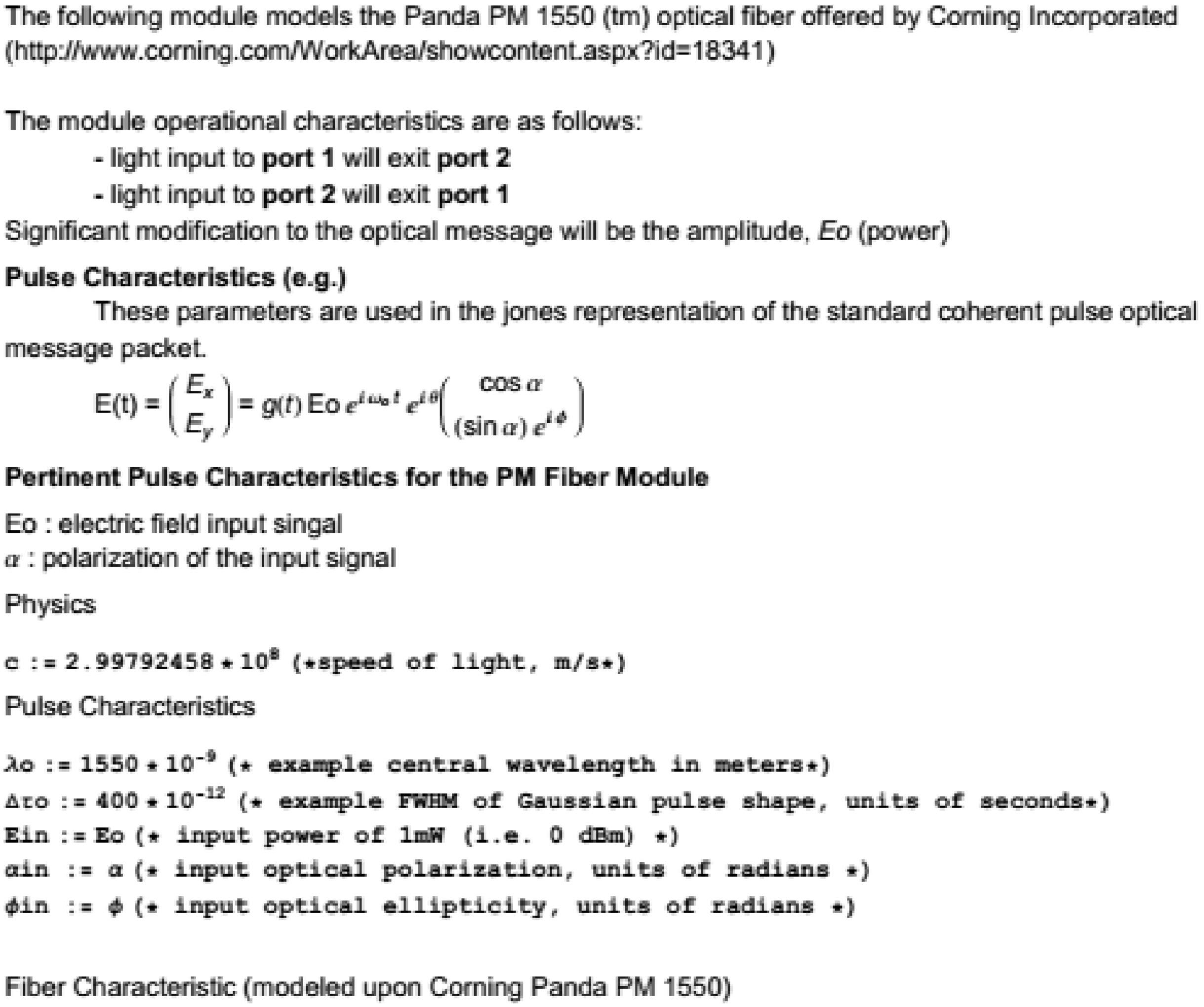
```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
   (output.port, output.pulse)
      if phase = “respond”

   ∅ (null output)
     otherwise;
```

Time advance Function:

ta(*phase*, σ, *store, temperature*, *overtemp*, *overpower, interruptRespond, queue*) = σ;

N.8 Mathematical Model

Pulse propagation considerations for the Polarization-Maintaining Fiber Module within the QKD OMNeT++ simulation environment



The following module models the Panda PM 1550 (tm) optical fiber offered by Corning Incorporated (http://www.corning.com/WorkArea/showcontent.aspx?id=18341)

The module operational characteristics are as follows:

- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

Significant modification to the optical message will be the amplitude, *Eo* (power)

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\omega_o t}\,e^{i\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the PM Fiber Module

Eo : electric field input singal

α : polarization of the input signal

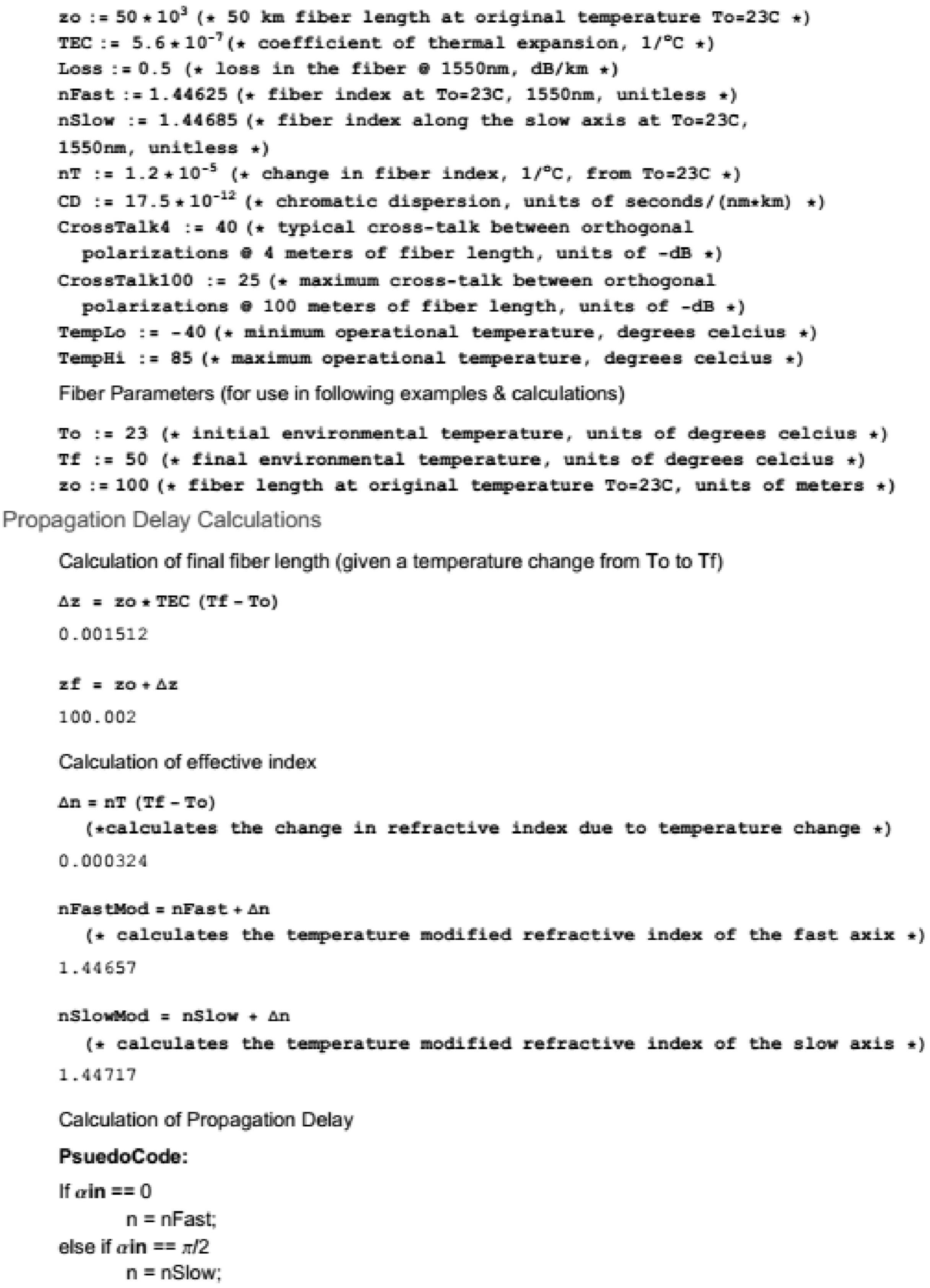
Physics

```
c := 2.99792458 * 10^8 (*speed of light, m/s*)
```

Pulse Characteristics

```
λo := 1550 * 10^-9 (* example central wavelength in meters*)
Δτo := 400 * 10^-12 (* example FWHM of Gaussian pulse shape, units of seconds*)
Ein := Eo (* input power of 1mW (i.e. 0 dBm) *)
αin := α (* input optical polarization, units of radians *)
ϕin := ϕ (* input optical ellipticity, units of radians *)
```

Fiber Characteristic (modeled upon Corning Panda PM 1550)



```
zo := 50 * 10^3 (* 50 km fiber length at original temperature To=23C *)
TEC := 5.6 * 10^-7 (* coefficient of thermal expansion, 1/°C *)
Loss := 0.5 (* loss in the fiber @ 1550nm, dB/km *)
nFast := 1.44625 (* fiber index at To=23C, 1550nm, unitless *)
nSlow := 1.44685 (* fiber index along the slow axis at To=23C,
1550nm, unitless *)
nT := 1.2 * 10^-5 (* change in fiber index, 1/°C, from To=23C *)
CD := 17.5 * 10^-12 (* chromatic dispersion, units of seconds/(nm*km) *)
CrossTalk4 := 40 (* typical cross-talk between orthogonal
  polarizations @ 4 meters of fiber length, units of -dB *)
CrossTalk100 := 25 (* maximum cross-talk between orthogonal
  polarizations @ 100 meters of fiber length, units of -dB *)
TempLo := -40 (* minimum operational temperature, degrees celcius *)
TempHi := 85 (* maximum operational temperature, degrees celcius *)
```

Fiber Parameters (for use in following examples & calculations)

```
To := 23 (* initial environmental temperature, units of degrees celcius *)
Tf := 50 (* final environmental temperature, units of degrees celcius *)
zo := 100 (* fiber length at original temperature To=23C, units of meters *)
```

Propagation Delay Calculations

Calculation of final fiber length (given a temperature change from To to Tf)

```
Δz = zo * TEC (Tf - To)
0.001512

zf = zo + Δz
100.002
```

Calculation of effective index

```
Δn = nT (Tf - To)
  (*calculates the change in refractive index due to temperature change *)
0.000324

nFastMod = nFast + Δn
  (* calculates the temperature modified refractive index of the fast axix *)
1.44657

nSlowMod = nSlow + Δn
  (* calculates the temperature modified refractive index of the slow axis *)
1.44717
```

Calculation of Propagation Delay

PsuedoCode:

```
If αin == 0
        n = nFast;
else if αin == π/2
        n = nSlow;
```

```
else
        error message : "improper injection to PM fiber"
end;
```

Assuming n = nFast

`n := nFastMod`

PropDelay = $\frac{zf * n}{c}$ (*final time is in seconds*)

4.82532×10^{-7}

Assuming n = nSlow

`n := nSlowMod`

PropDelay = $\frac{zf * n}{c}$ (*final time is in seconds*)

4.82733×10^{-7}

The same thing, but in long form (assuming α**in=0**, thus n = nFast)

PropDelay = $\frac{zo}{c}$ (1 + TEC (Tf - To)) (nFast + nT (Tf - To))

4.82532×10^{-7}

Chromatic Dispersion (pulse broadening) Calculations (not required for current model)

Note: Due to the very small effect this has on our system due to the a.) lengths of fiber being used and b.) the (relatively) long temporal duration and corresponding narrow spectral width, this effect need not be included in the initial versions of the OMNet++ module.

For this example we assume the time-power profile of the pulse is a bandwidth-limited "ideal" Gaussian. Thus, the time-frequency bandwidth product is; $\Delta\tau o$*$\Delta\nu o$ = 0.441. We need to calculate the spread in wavelength in the pulse to calcuate the pulse broadening

Calculation of original spectral (frequency) spread

Δvo = 0.441 / Δτo
  (*calculates the frequency FWHM for the original pulse, units of Hertz*)

1.1025×10^{9}

vo = c / λo // N
  (*calculates center frequency given central wavelength λo, units of Hertz*)

1.93414×10^{14}

Δλ = $\frac{\Delta vo * (\lambda o)^2}{c}$ (*calculates the spectral FWHM, units in meters*)

8.8353×10^{-12}

As can be seen, $\Delta\lambda$ is very small, so we should expect little pulse broadening due to chromatic dispersion.

$$\Delta\tau\mathrm{CD} = \mathrm{CD} * \frac{\Delta\lambda}{10^{-9}} * \frac{\mathrm{zf}}{1000}\ \texttt{(* spread due to chromatic dispersion, units of seconds *)}$$

1.5462×10^{-14}

$$\Delta\tau\mathrm{f} = \sqrt{(\Delta\tau\mathrm{o})^2 + (\Delta\tau\mathrm{CD})^2}$$

```
(*final pulse duration,
 units of seconds.  Note that the spread is so small over 100m that its effect
  is negligible when compared to the duration of the original pulse *)
```

$4. \times 10^{-10}$

The same thing, but in long form

$$\Delta\tau\mathrm{f2} = \sqrt{(\Delta\tau\mathrm{o})^2 + \left(\mathrm{CD} * \frac{1}{10^{-9}}\left(\frac{0.441 / \Delta\tau\mathrm{o} * (\lambda\mathrm{o})^2}{\mathrm{c}}\right) * \frac{\mathrm{zf}}{1000}\right)^2}$$

$4. \times 10^{-10}$

Crosstalk Calculations (not required for current model)

This may be used, if desired. Like the Chromatic Dispersion calculation, the effect is vanishingly small over short distances, so we may not wish to include this effect in the inital version of the module

Note that the crosstalk can be estimated as being linear in dB for fiber above a few meters in length. For lengths near the crosstalk parameters given we can extrapolate the effective crosstalk. Note that crosstalk can be cyclic and is effected by loss in the fiber, requiring a different, more accurate model for extremely long fiber lengths.

$$\texttt{CrossTalkLevel[FiberLength\_, CT1\_, z1\_, CT2\_, z2\_] :=}$$
$$\left(\frac{\mathrm{CT2} - \mathrm{CT1}}{\mathrm{z2} - \mathrm{z1}}\right) * \mathrm{FiberLength} + \left(\mathrm{CT1} - \left(\frac{\mathrm{CT2} - \mathrm{CT1}}{\mathrm{z2} - \mathrm{z1}}\right)\mathrm{z1}\right)$$

Calculation of the crosstalk level after transiting the fiber

```
CrossTalkLevel[zf, CrossTalk4, 4, CrossTalk100, 100] //
 N (* Optical power of crosstalk signal, units of -dB *)
```

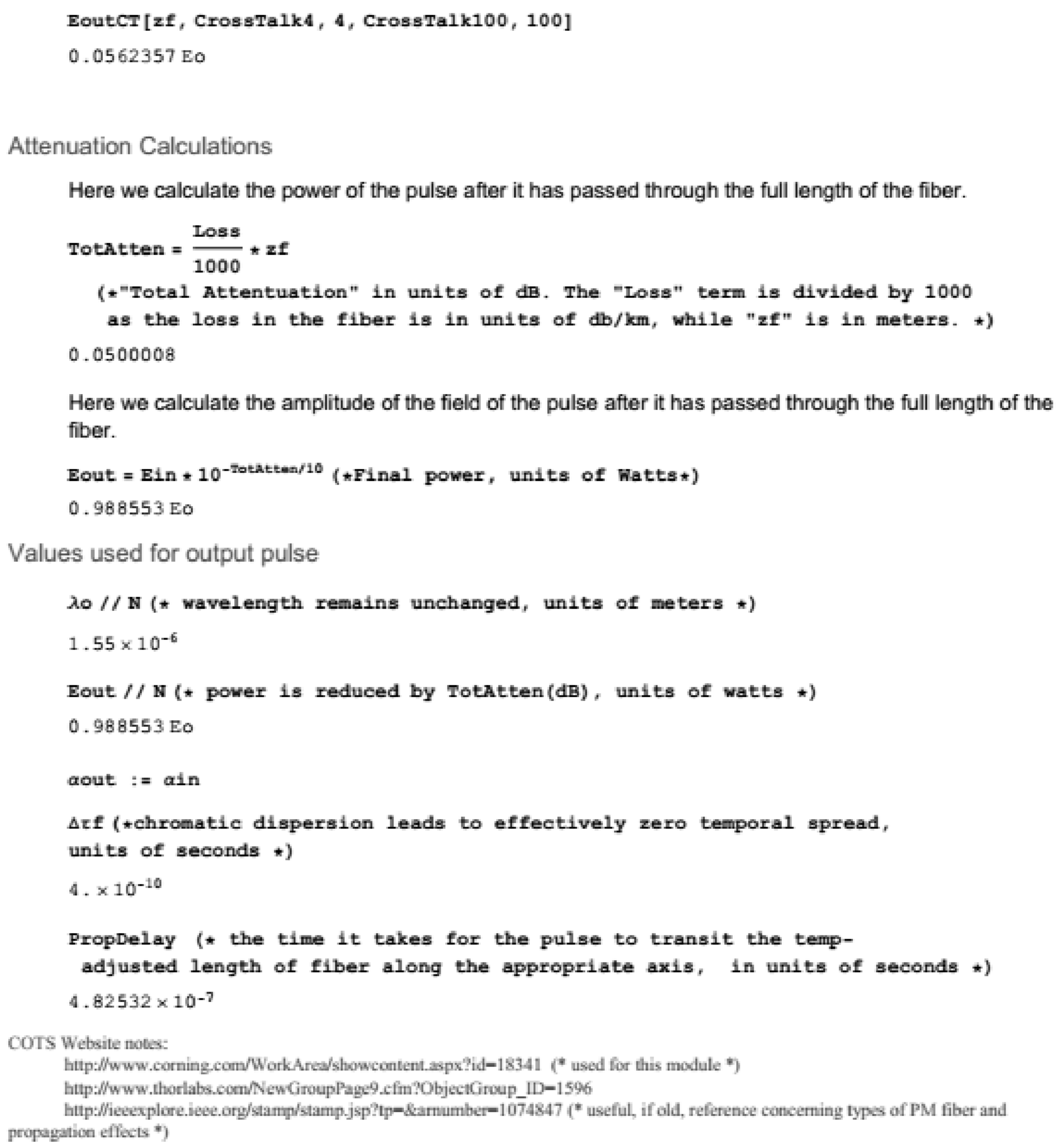
```
24.9998
```

to calculate the undesired crosstalk output PsuedoCode:

```
If $\alpha$in == 0
        $\alpha$outCT = $\pi/2$;
else if $\alpha$in == $\pi/2$
        $\alpha$outCT = 0;
else
        error message : "improper injection to PM fiber"
end;
```

Calculating the power of the crosstalk output:

$$\texttt{EoutCT[FiberLen\_, CT1\_, z1\_, CT2\_, z2\_] :=}\ \mathrm{Ein} * \sqrt{10^{-\mathrm{CrossTalkLevel[FiberLen,\ CT1,z1,CT2,z2]/10}}}$$



```
EoutCT[zf, CrossTalk4, 4, CrossTalk100, 100]
0.0562357 Eo
```

Attenuation Calculations

Here we calculate the power of the pulse after it has passed through the full length of the fiber.

$$\text{TotAtten} = \frac{\text{Loss}}{1000} * \text{zf}$$

```
  (*"Total Attentuation" in units of dB. The "Loss" term is divided by 1000
   as the loss in the fiber is in units of db/km, while "zf" is in meters. *)
0.0500008
```

Here we calculate the amplitude of the field of the pulse after it has passed through the full length of the fiber.

$\text{Eout} = \text{Ein} * 10^{-\text{TotAtten}/10}$ `(*Final power, units of Watts*)`

```
0.988553 Eo
```

Values used for output pulse

```
λo // N (* wavelength remains unchanged, units of meters *)
```

1.55×10^{-6}

```
Eout // N (* power is reduced by TotAtten(dB), units of watts *)
0.988553 Eo

αout := αin

Δτf (*chromatic dispersion leads to effectively zero temporal spread,
units of seconds *)
```

$4. \times 10^{-10}$

```
PropDelay (* the time it takes for the pulse to transit the temp-
 adjusted length of fiber along the appropriate axis,  in units of seconds *)
```

4.82532×10^{-7}

COTS Website notes:

http://www.corning.com/WorkArea/showcontent.aspx?id=18341 (* used for this module *)

http://www.thorlabs.com/NewGroupPage9.cfm?ObjectGroup_ID=1596

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1074847 (* useful, if old, reference concerning types of PM fiber and propagation effects *)

N.9 Component Use Case

N.9.1 Respond to an Optical Packet in the PM Fiber

Optical packet arrives at the PM fiber. Place the optical packet into the optical queue. Check to see if optical packet overpowers the PM fiber. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated optical output signal based on the input signal, length and type of fiber, and the current component state. Maintain the optical packet polarization. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
  - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state

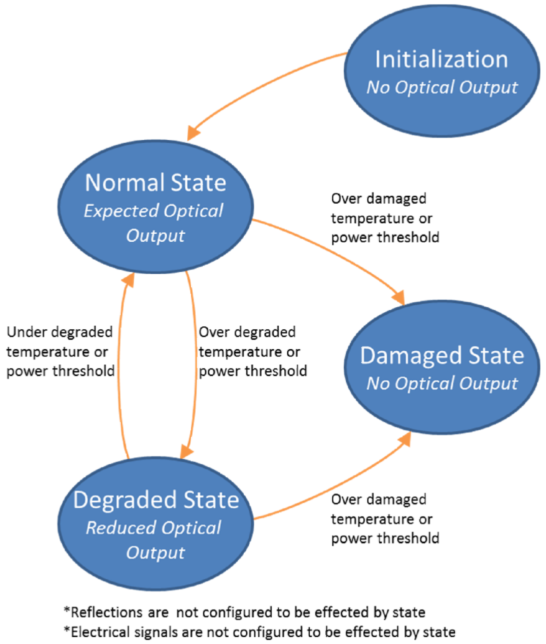

Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold
*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 134. Component states.

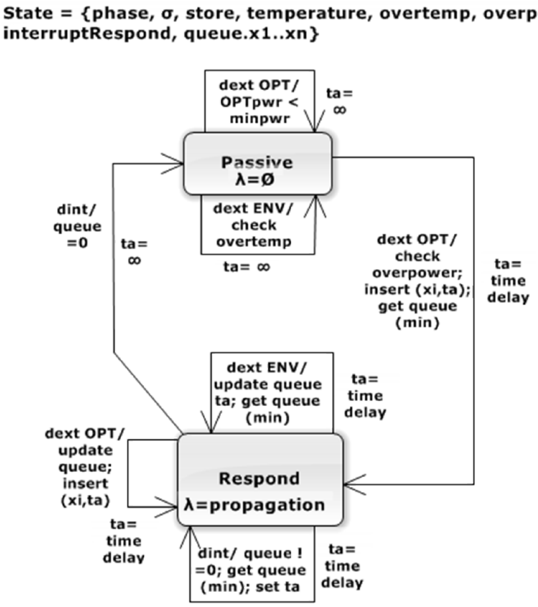

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
dext OPT/ OPTpwr < minpwr
ta= ∞
Passive λ=Ø
dext ENV/ check overtemp
ta= ∞
dint/ queue =0
ta= ∞
dext OPT/ check overpower; insert (xi,ta); get queue (min)
ta= time delay
dext ENV/ update queue ta; get queue (min)
ta= time delay
dext OPT/ update queue; insert (xi,ta)
ta= time delay
Respond λ=propagation
dint/ queue ! =0; get queue (min); set ta
ta= time delay

Figure 135. PM fiber phase transition diagram.

N.9.2 Respond to Optical Packet End Goals

- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Optical packet attenuated properly to the limit of accuracy
- Optical packet propagated out the correct port at the correct time

N.9.3 Respond to an Environmental Packet in the PM Fiber

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

N.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

N.10 PM Fiber Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *PM Fiber Test Cases*

		Inject Ports				Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Env**	**Notes**	**opt #**	**env #**
Passive	1	1	0	0	single	1	0
	2	0	1	0	single	2	0
	3	0	0	1	single	2	1
	4	1	1	0	same time	4	1
	5	1	1	0	differ time	6	1
	6	1	1	1	same time	8	2
	7	1	1	1	differ time	10	3
	8	0	1	1	same time	11	4
	9	0	1	1	differ time	12	5
	10	1	0	1	same time	13	6
	11	1	0	1	differ time	14	7
	20	2	0	0	same time	16	7
	21	0	2	0	same time	18	7
	22	2	2	0	same time	22	7
	23	2	2	0	differ time	26	7
	24	2	2	1	same time	30	8
	25	2	2	1	differ time	34	9
	26	0	2	1	same time	36	10
	27	0	2	1	differ time	38	11
	28	2	0	1	same time	40	12
	29	2	0	1	differ time	42	13
totals		21	21	13	42		
Respond	41	2	0	0	single	44	13
	42	0	2	0	single	46	13
	43	1	0	1	single	47	14
	44	2	1	0	same time	50	14
	45	2	1	0	differ time	53	14
	46	2	1	1	same time	56	15
	47	2	1	1	differ time	59	16
	48	0	2	1	same time	61	17
	49	0	2	1	differ time	63	18
	50	2	0	1	same time	65	19
	51	2	0	1	differ time	67	20
	60	3	0	0	same time	70	20
	61	0	3	0	same time	73	20
	62	3	2	0	same time	78	20
	63	3	2	0	differ time	83	20
	64	3	2	1	same time	88	21
	65	3	2	1	differ time	93	22

66	0	3	1	same time	96	23
67	0	3	1	differ time	99	24
68	3	0	1	same time	102	25
69	3	0	1	differ time	105	26
totals	36	27	13	63		
TC1	1	0	2	single	106	28
TC2	1	0	2	single	107	30
TC3	1	0	2	single	108	32
TC4	1	0	2	single	109	34
TC5	1	0	2	single	110	36
TC6	1	0	2	single	111	38
TC7	1	0	2	single	112	40
totals	7	0	14	7		

Notes: 5 - under minimum power packet sent on OPT1; OPT1 & OPT2 - differ time - Passive
7 - under minimum power packet sent on OPT2; OPT1, OPT2, ENV - differ time - Passive

N.11 References

Corning. (2013). Panda pm speciality optical fibers. Retrieved October 8, 2013, from http://www.corning.com/WorkArea/showcontent.aspx?id=18341

OZOptics. (2014). Accurate alignment preserves polarization
. Retrieved March 31, 2014, from http://www.ozoptics.com/ALLNEW_PDF/ART0001.pdf

RPPhotonics. (2013). RP phontics encylopedia - polarization-maintaining fibers
. Retrieved March 31, 2014, from http://www.rp-photonics.com/polarization_maintaining_fibers.html

Appendix O - Polarization Controller (Deterministic)

O.1 Device Description:

The polarization controller (PC) is a device that changes the polarization state of light that passes through it, converting light from any random polarization state into a specific output polarization state. A deterministic controller is computer-controlled to output the light automatically without needing operator inputs during operation. The device consists of polarimeter and a state of polarization (SOP) controller combined with a computer and software. See Figure 1 for an example of a deterministic polarization controller.

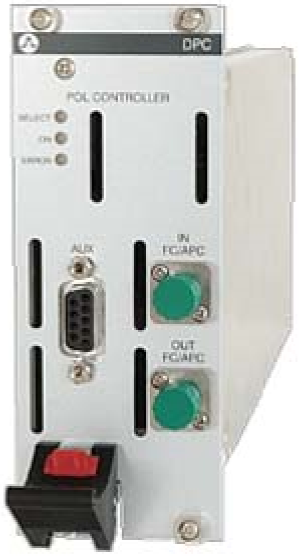


Figure 136. View of a deterministic polarization controller (ThorLabs, 2013).

The Polarization controller is a bidirectional optical component with two optical ports and computer port. Optical signals arriving at the input port are propagated to the other port after a defined propagation delay and the polarization is changed to a specific point on the Poincare sphere and then output through the other port. The polarizing system is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the polarization controller may become permanently damaged which changes its propagation characteristics. Similarly, the Polarization controller is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the Polarization controller may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the polarization controller is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the Polarization controller. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the Polarization controller to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the polarization controller using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the polarization controller. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

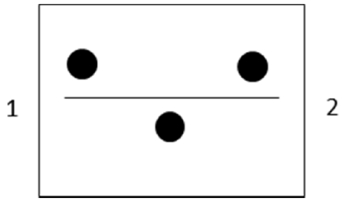


Figure 137. Symbol for the polarization controller in the QKD system architecture.

O.2 Polarization Controller Conceptual Model

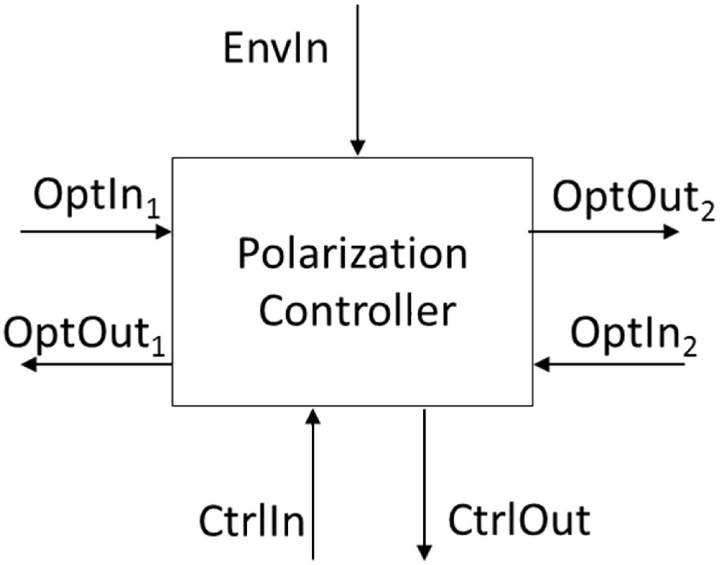


Figure 138. Polarization controller conceptual model.

The conceptual model for a polarization controller consists of two optical input ports {$OptIn_1$, $OptIn_2$}, two optical output ports {$OptOut_1$, $OptOut_2$}, one environmental input port

{EvnIn} and one electrical controller input port and one electrical controller output port {CtrlIn, CtrlOut}. The environmental port allows external sources to communicate changes in the operational environment to the polarization controller. The electrical controller ports allow for control inputs to the controller and responses from the polarization controller to the higher system functions.

In comparison to the polarization controller symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. The electrical control port is not shown for clarity in Figure 2, and is also decomposed in the model into an input port and an output port. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the polarization controller, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 3 will instantaneously generate a reflected emitting out of OptOut_1.

The polarization controller calculates changes to the amplitude, and polarization ellipticity and orientation of any packet coming through either optical port after a time equaling the propagation delay of the module. The polarization controller determines the input optical packet polarization and changes the polarization to match the output polarization set by the control messages for packets entering port 1 moving in the forward direction. Packets entering port 2 and moving in the reverse direction experience a change to the current polarization opposite to that from those entering port 1. Sufficient time-averaged optical power is required for the polarization controller to operate as the controller has a minimum optical power level threshold necessary to make changes.

In the model, every quantum-level pulse is changed by the current polarization controller settings, but only bright pulses cause an update to the current polarization settings. The bright pulses in our model provide the sufficient time-average power to enable the controller to make polarization changes. Every output packet is calculated at full power minus some small amount to account for attenuation (insertion loss and Polarization Dependent Loss) through the device and the ellipticity and orientation are changed to match the required output ellipticity and orientation.

The polarization controller must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the Polarization controller can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

O.3 Mathematical Model

For a detailed mathematical description of the Polarization controller, refer to Section 13.8 which contains the Mathematica worksheet provided by the optical physics SME.

O.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the Polarization controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue
- Calculate the reflected power of the signal and send its output with the same port number.
- Retrieve the input optical signal from the queue, and calculate the attenuated output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment
- Update the values of the input optical signal based on the characteristics of the controller, the original values of the input optical signal and the current environment.
- Send the changed output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

When a control message arrives:

- Update the *αset* and *ϕset* with the values in the control message

O.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the Polarization controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the Polarization controller at the same time.

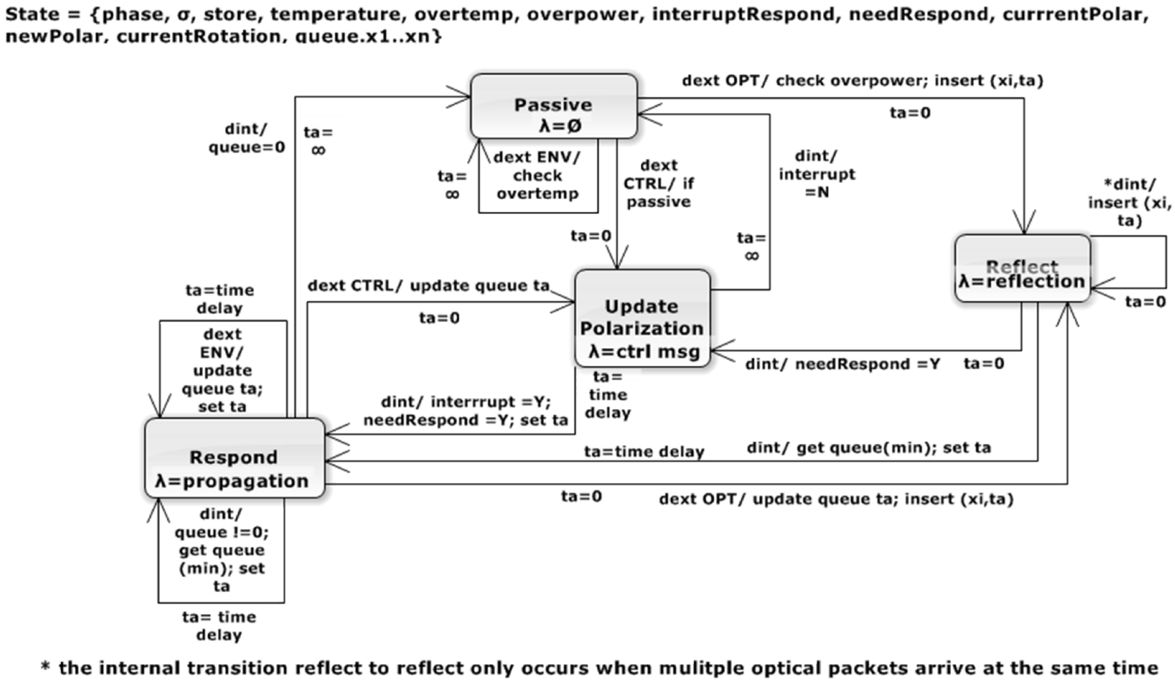

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, currrentPolar, newPolar, currentRotation, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dint/ queue=0
ta= ∞
ta= ∞
dext ENV/ check overtemp
dext CTRL/ if passive
ta=0
dint/ interrupt =N
ta= ∞
*dint/ insert (xi, ta)
Reflect
λ=reflection
ta=0
ta=time delay
dext CTRL/ update queue ta
ta=0
Update Polarization
λ=ctrl msg
dint/ needRespond =Y
ta=0
dext ENV/ update queue ta; set ta
ta= time delay
dint/ interrrupt =Y; needRespond =Y; set ta
ta=time delay
dint/ get queue(min); set ta
Respond
λ=propagation
ta=0
dext OPT/ update queue ta; insert (xi,ta)
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 139. Polarization controller phase transition diagram.

O.6 Event-Trace Diagram

This section shows various examples of packets entering the Polarization controller. The tables list the states the polarization controller proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the Polarization controller, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

O.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 56. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**curr polar**	**queue (xi, *tp*)**	**Notes: assume tp= 5**
	1-packet	no env	no ext	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	*ϕ, α*	null	
0	s0	exit	passive	0	null	c	n	n	n	n	*ϕ, α*	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	*ϕ, α*	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	*ϕ, α*	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	*ϕ, α*	null	
5	s2	exit	respond	inf	x1	c	n	n	n	n	*ϕ, α*	null	
5	s3	entry	passive	inf	x1	c	n	n	n	n	*ϕ, α*	null	

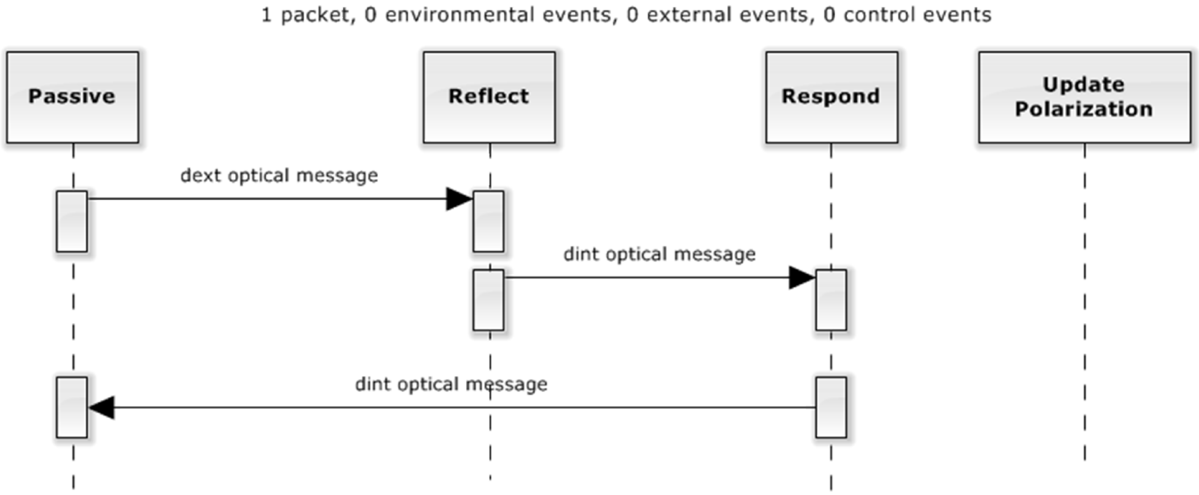


Figure 140. Case I sequence diagram.

O.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 57. *Case II state list*.

Time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	0 env	1 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	ϕ, α	null	
0	s0	exit	passive	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	ϕ, α	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	ϕ, α	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	ϕ, α	(x2,5)	dext at e=2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	ϕ, α	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	ϕ, α	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	ϕ, α	(x2,5)	
5	s4	exit	respond	2	x2	c	n	n	n	n	ϕ, α	null	
5	s5	entry	respond	2	x2	c	n	n	n	n	ϕ, α	null	
7	s5	exit	respond	inf	x2	c	n	n	n	n	ϕ, α	null	
7	s6	entry	passive	inf	x2	c	n	n	n	n	ϕ, α	null	

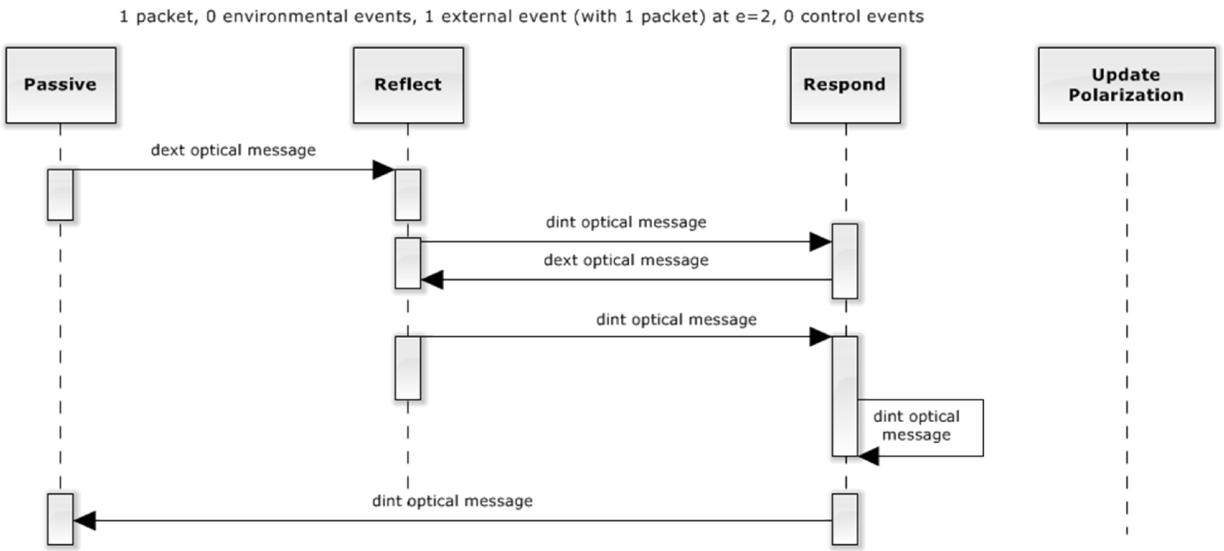


Figure 141. Case II sequence diagram.

O.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 58. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	0 env	2 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	ϕ, α	null	
0	s0	exit	passive	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	ϕ, α	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	ϕ, α	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	ϕ, α	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	ϕ, α	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	ϕ, α	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	ϕ, α	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	n	θ	(x2,4) (x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	n	ϕ, α	(x2,4) (x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	n	ϕ, α	(x2,4) (x3,5) (x4,5)	

3	s6	entry	reflect	0	x1	c	n	n	y	n	ϕ, α	(x2,4) (x3,5) (x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	n	ϕ, α	(x2,4) (x3,5) (x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	n	ϕ, α	(x2,4) (x3,5) (x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	n	ϕ, α	(x3,2) (x4,2)	
5	s8	entry	respond	2	x2	c	n	n	n	n	ϕ, α	(x3,2) (x4,2)	
7	s8	exit	respond	1	x3	c	n	n	n	n	ϕ, α	(x4,0)	
7	s9	entry	respond	1	x3	c	n	n	n	n	ϕ, α	(x4,0)	
8	s9	exit	respond	0	x4	c	n	n	n	n	ϕ, α	null	
8	s10	entry	respond	0	x4	c	n	n	n	n	ϕ, α	null	
8	s10	exit	respond	inf	x4	c	n	n	n	n	ϕ, α	null	
8	s11	entry	passive	inf	x4	c	n	n	n	n	ϕ, α	null	

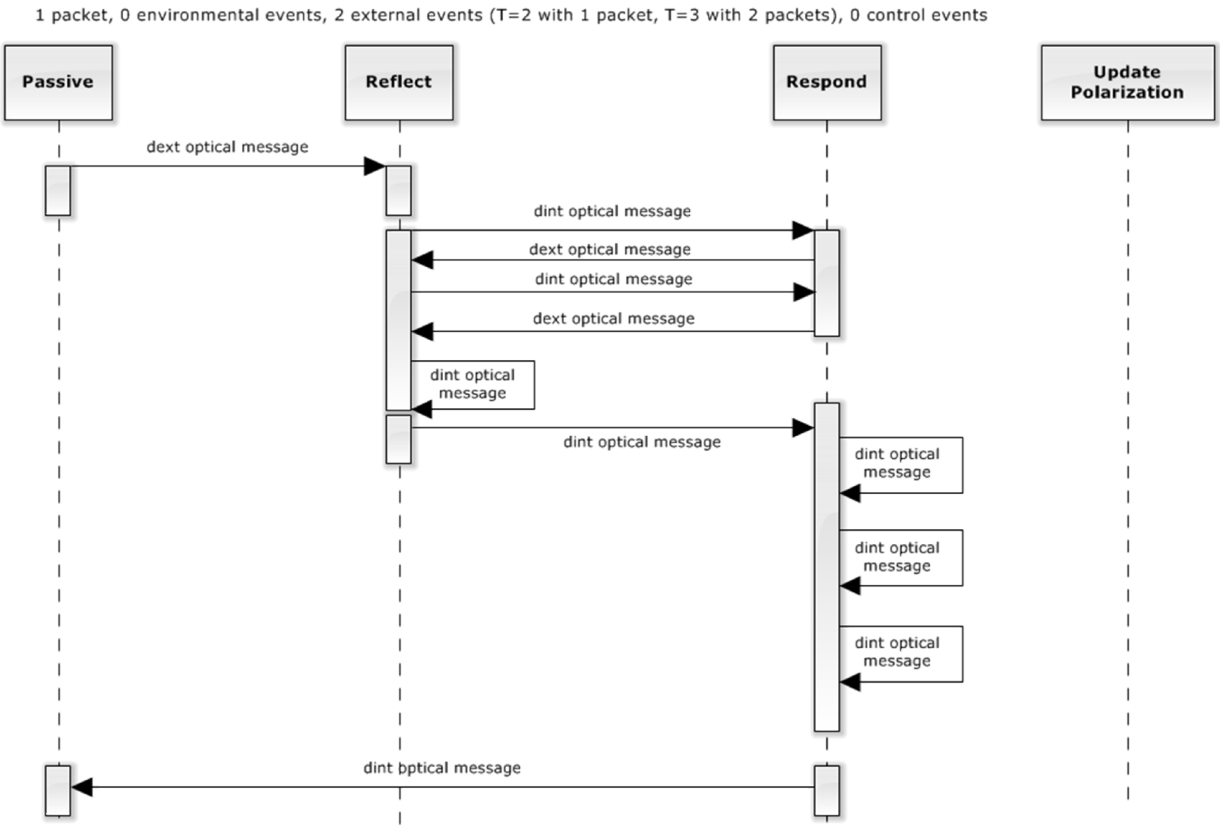


Figure 142. Case III sequence diagram.

O.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 59. *Case IV state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	1 env	0 ext	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	ϕ, α	null	
0	s0	exit	passive	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	ϕ, α	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	ϕ, α	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	y	n	ϕ, α	null	update temp
3	s3	entry	respond	2	x1	c	y	n	y	n	ϕ, α	null	
5	s3	exit	respond	inf	x1	c2	y	n	n	n	ϕ, α	null	
5	s4	entry	passive	inf	x1	c2	y	n	n	n	ϕ, α	null	

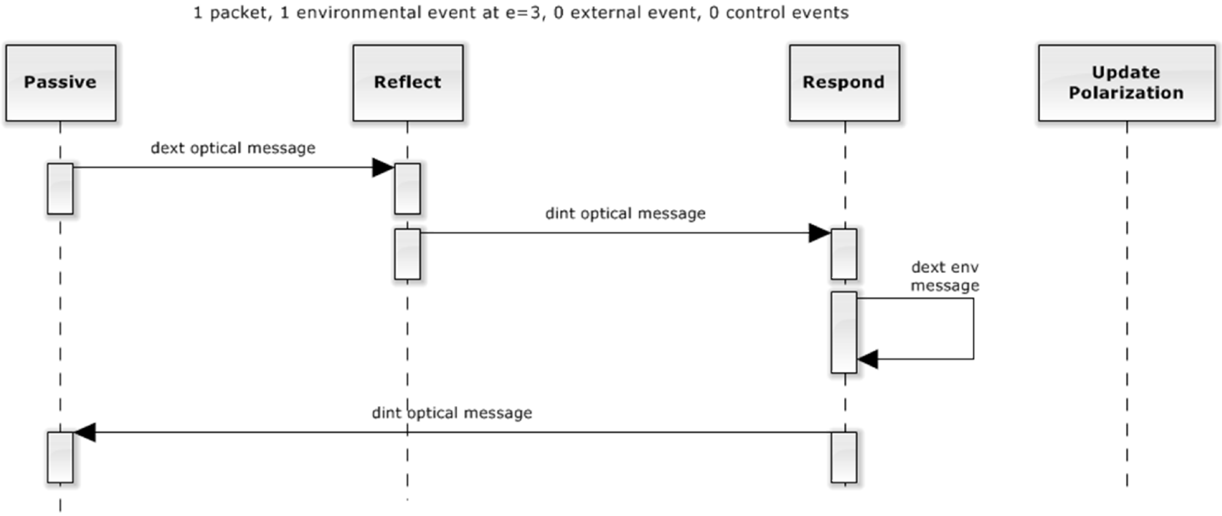


Figure 143. Case IV sequence diagram.

O.6.5 CASE V: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Control Packet Arriving at Time 3

Table 60. *Case V state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1 opt	1 env	0 opt	1 ctrl									

0	s0	entry	passive	inf	null	c	n	n	n	n	ϕ, α	null	
0	s0	exit	passive	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	ϕ, α	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	ϕ, α	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	ϕ, α	(x1,5)	CTRL arrives e=3
3	s2	exit	respond	0	x1	c	n	n	y	n	ϕ, α	(x1,2)	
3	s3	entry	update detector	0	x1	c	n	n	y	n	ϕ, α	(x1,2)	
3	s3	exit	update detector	2	x1	c	n	n	y	n	ϕ, α	(x1,2)	
3	s4	entry	respond	2	x1	c	n	n	y	n	$\phi +, \alpha$	(x1,2)	
5	s4	exit	respond	0	x1	c	n	n	n	n	$\phi +, \alpha$	null	
5	s6	entry	passive	inf	x1	c	n	n	n	n	$\phi +, \alpha$	null	

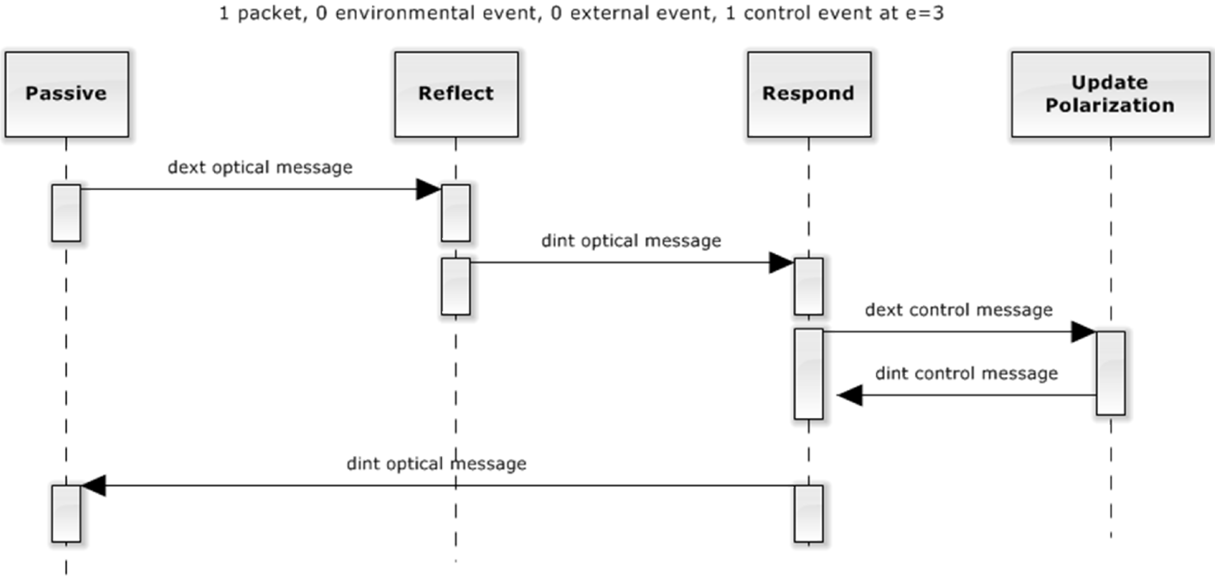


Figure 144. Case V sequence diagram.

O.7 Polarization controller Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- Assume that only one control packet will arrive at any given time, due to the small time scales involved and the length of time necessary for polarization changes.

- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, ”interruptRespond”, “needRespond”, “currentPolar”, “newPolar”, “currentRotation”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the polarization controller we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

$\boldsymbol{DEVS_{Polarization\ controller} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “reflect”, “respond”, “update detector”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
needRespond= flag variable set when both Reflect and UpdateDetector respond to inputs
currentPolar = current polarization and ellipticity values of the controller
newPolar = polarization the controller is changing to after receiving a change message
currentRotation = polarization and ellipticity rotation values for the current packet
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
ctrlOutput = variable that stores the output control message response
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
minSet = minimum optical pulse power necessary for a change to output polarization
maxSet = maximum optical pulse power allowed for a change to output polarization
αset = fixed output orientation of the polarizer, set by user
ϕset = fixed output ellipticity of the polarizer, set by user
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
ctrlMsg() = method that generates a response message to received control messages
outputMsg() = method that generates the response message to received optical packets
insert_event_q() = method that inserts the current (x_i, time delay_i) into the queue

remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr, currentPolar, newPolar, currentRotation*)
calcPolarSet() = method that calculates the current polarization setting as *f*(*αset,current.v*)
calcReflected() = method that calculates reflection power of an optical packet
changePolarization() = method that changes current polarization of the controller
calcAttenPolar() = method that calculates the optical output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr, currentPolarization*)
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
$q.v_{min}$ = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"$OptIn_1$", "$OptIn_2$", "EnvIn", "CtrlIn"} with
X_M = {("$OptIn_1$", V_{opt}), ("$OptIn_2$", V_{opt}), ("EnvIn", V_{env}), ("CtrlIn", V_{ctrl})} is the set of input ports and values.

OutPorts = {"$OptOut_1$", "$OptOut_2$", "CtrlOut"} with
Y_M = {("$OptOut_1$", $Y_{OptOut1}$), ("$OptOut_2$", $Y_{OptOut2}$), ("CtrlOut", $Y_{CtrlOut}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower*, *interruptRespond, needRespond, currentPolar, newPolar, currentRotation, queue*} = {{"passive", "reflect", "respond", "update polarization"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x {"Y","N"} x {"Y","N"} x *V* x *V* x *V* x *V*}

External Transition Function:

δ_{ext}(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond , currentPolar, newPolar, currentRotation, queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =

```
(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                          currentPolar, newPolar, currentRotation, queue.x1..xn)
  if phase = “passive” and p ∈ {“OptIn1”, “OptIn2”}
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower =  “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”
  if currentPolar != newPolar
      currentPolar = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr,
currentPolar, newPolar, currentRotation)

 (“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                          currentPolar, newPolar, currentRotation, queue.x1..xn)
  if phase = “respond” and p ∈ {“OptIn1”, “OptIn2”}
    update_delay(queue)
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower =  “Y”
      insert_event_q(current)
      remove_event_m(current)
   queue.current = queue_need_reflected()
   reflect = (queue.current.p), calcReflected(queue.current.v))
   mark_reflected(queue.current)
   interruptRespond= “Y”
   if currentPolar != newPolar
    currentPolar = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr,
currentPolar, newPolar, currentRotation)
  timeLeftRespond = timeLeftRespond - e

(“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                          currentPolar, newPolar, currentRotation, queue.x1..xn)
   if phase = “passive” and p = “EnvIn”
   temperature = messagebag.temperature
   if temperature > damage.temp
```

```
      overtemp = “Y”
    if currentPolar != newPolar
     currentPolar = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr,
currentPolar, newPolar, currentRotation)

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                          needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
   if phase = “respond” and p = “EnvIn”
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”
    if currentPolar != newPolar
     currentPolar = calcPolar(current.v, temperature, overtemp, peakpwr, overpwr,
currentPolar, newPolar, currentRotation)
     time.delay = timeLeftRespond

(“update polarization”, 0, store, temperature, overtemp, overpower, interruptRespond,
                          needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
   if phase = “passive” and p =“CtrlIn”
     ctrlOutput = ctrlMsg(store)
     newPolar = store.value.polarization

(“update polarization”, 0, store, temperature, overtemp, overpower, interruptRespond,
                          needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
   if phase = “respond” and p = “CtrlIn”
     update_delay(queue)
     ctrlOutput = ctrlMsg(store)
     interruptRespond= “Y”
     newPolar = store.value.polarization

(phase, $\sigma - e$, ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                    currentPolar, newPolar, currentRotation, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                              currentPolar, newPolar, currentRotation, queue)=
(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                    currentPolar, newPolar, currentRotation, queue.x1..xn))
  if phase = “reflect” and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
    if current.value.power > MinSet and < MaxSet
```

```
         newPolar = calcPolarSet(current)
        reflect = (current.p), calcReflected(current.v))
        mark_reflected(current)

    (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                        needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
       if phase = “reflect” and need.reflect = null
        need.reflect = queue_need_reflected()
        if interruptRespond = “N”
          current = queue_min()
          time.delay = current.time.delay
          if current.value.power > MinSet and < MaxSet
           newPolar = calcPolarSet(current)
         if InPort = “OptIn$_1$”
           outputPulse = calcPolar(store.v, temperature, overtemp, peakpwr, overpwr, currentPolar,
    newPolar, currentRotation)
           outputPort = “OptOut$_2$”
         if InPort = “OptIn$_2$”
             outputPulse = calcPolar(store.v, temperature, overtemp, peakpwr, overpwr, currentPolar,
    newPolar, currentRotation)
           outputPort = “OptOut$_1$”
         timeLeftRespond = propagation delay
       else
         time.delay = timeLeftRespond

(“update polarization”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                        currentPolar, newPolar, currentRotation, queue.x1..xn)
        if phase = “reflect” and needRespond = “Y”
         ctrlOutput = ctrlMsg(store)

     (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                        needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
        if phase = “respond” and size > 0
         update_delay(queue)
         size= queue_size()
         current = queue_min()
         time.delay = current.time.delay
          if current.value.power > MinSet and < MaxSet
           newPolar = calcPolarSet(current)
         if InPort = “OptIn$_1$”
             outputPulse = calcPolar(store.v, temperature, overtemp, peakpwr, overpwr, currentPolar,
    newPolar, currentRotation)
           outputPort = “OptOut$_2$”
         if InPort = “OptIn$_2$”
             outputPulse = calcPolar(store.v, temperature, overtemp, peakpwr, overpwr, currentPolar,
    newPolar, currentRotation)
```

```
      outputPort = “OptOut$_1$”
    interruptRespond= “N”

  (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                currentPolar, newPolar, currentRotation, queue.x1..xn)
    if phase = “respond” and size  = 0
      size= queue_size()


  (“passive”, $\infty$, store, temperature, overtemp, overpower, overpower, interruptRespond,
                      needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
    if phase = “update polarization” and interruptRespond = “N”


  (“respond”, time.delay, store, temperature, overtemp, overpower, overpower, interruptRespond,
                      needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
    if phase = “update polarization” and interruptRespond = “Y”
      time.delay = timeLeftRespond

  (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                      needRespond, currentPolar, newPolar, currentRotation, queue.x1..xn)
    if phase = “update polarization” and interruptRespond = “N” and needRespond = “Y”
      current = queue_min()
      time.delay = current.time.delay
        if current.value.power > MinSet and < MaxSet
         newPolar = calcPolarSet(current)
       if InPort = “OptIn$_1$”
          outputPulse = calcPolar(store.v, temperature, overtemp, peakpwr, overpwr, currentPolar,
newPolar, currentRotation)
        outputPort = “OptOut$_2$”
       if InPort = “OptIn$_2$”
          outputPulse = calcPolar(store.v, temperature, overtemp, peakpwr, overpwr, currentPolar,
newPolar, currentRotation)
        outputPort = “OptOut$_1$”
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                    currentPolar, newPolar, currentRotation, queue) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (outputPort, outputPulse)
```

if phase = “respond”

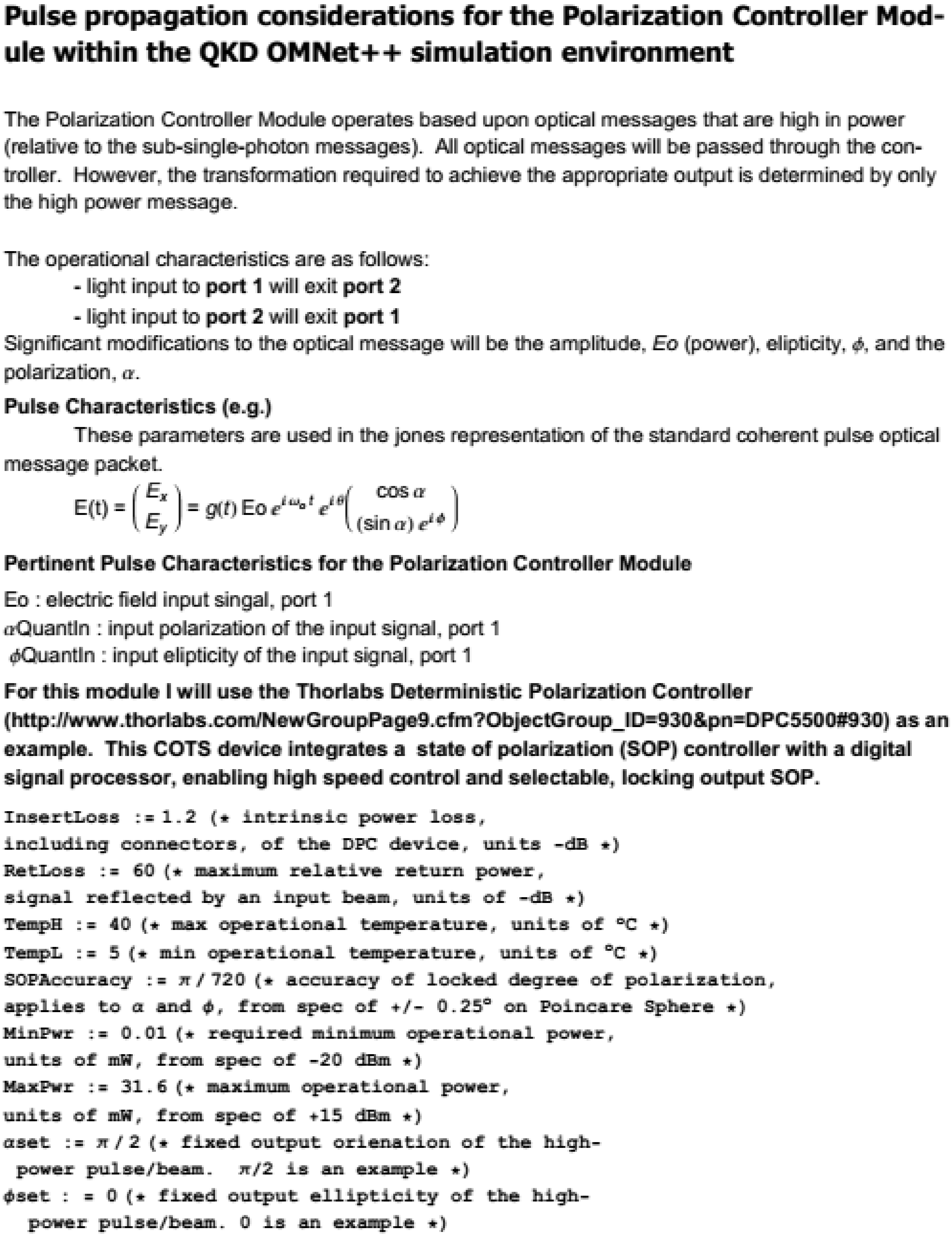
(“CtrlOut”, *ctrlOutput*)
if phase = “update polarization”

∅ (null output)
otherwise;

Time advance Function:

ta(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond, currentPolar, newPolar, currentRotation, queue*) = *σ*;

O.8 Mathematical Model



Pulse propagation considerations for the Polarization Controller Module within the QKD OMNet++ simulation environment

The Polarization Controller Module operates based upon optical messages that are high in power (relative to the sub-single-photon messages). All optical messages will be passed through the controller. However, the transformation required to achieve the appropriate output is determined by only the high power message.

The operational characteristics are as follows:

- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

Significant modifications to the optical message will be the amplitude, *Eo* (power), elipticity, ϕ, and the polarization, α.

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\, e^{i\,\omega_o\,t}\, e^{i\,\theta} \begin{pmatrix} \cos\alpha \\ (\sin\alpha)\, e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Polarization Controller Module

Eo : electric field input singal, port 1

αQuantIn : input polarization of the input signal, port 1

ϕQuantIn : input elipticity of the input signal, port 1

For this module I will use the Thorlabs Deterministic Polarization Controller (http://www.thorlabs.com/NewGroupPage9.cfm?ObjectGroup_ID=930&pn=DPC5500#930) as an example. This COTS device integrates a state of polarization (SOP) controller with a digital signal processor, enabling high speed control and selectable, locking output SOP.

```
InsertLoss := 1.2 (* intrinsic power loss,
including connectors, of the DPC device, units -dB *)
RetLoss := 60 (* maximum relative return power,
signal reflected by an input beam, units of -dB *)
TempH := 40 (* max operational temperature, units of °C *)
TempL := 5 (* min operational temperature, units of °C *)
SOPAccuracy := π / 720 (* accuracy of locked degree of polarization,
applies to α and ϕ, from spec of +/- 0.25° on Poincare Sphere *)
MinPwr := 0.01 (* required minimum operational power,
units of mW, from spec of -20 dBm *)
MaxPwr := 31.6 (* maximum operational power,
units of mW, from spec of +15 dBm *)
αset := π / 2 (* fixed output orienation of the high-
 power pulse/beam.  π/2 is an example *)
ϕset : = 0 (* fixed output ellipticity of the high-
  power pulse/beam. 0 is an example *)
```

Amplitude Attenuation Calculations for Polarization Controller (High-power and Quantum Level Signals)

To calculate the output amplitude we need to include insertion loss

```
AmplitudeIn := Eo
```

$$\texttt{Eout[AmplitudeIn\_, InsertLoss\_] = AmplitudeIn} * \sqrt{10^{-\text{InsertLoss}/10}}\ \texttt{// N}$$

```
0.870964 Eo
```

If we wish to flag the controller to include **undesired return (reflected)** messages, the following operations would hold true,

$$\texttt{EReturn[AmplitudeIn\_, RetLoss\_] = AmplitudeIn} * \sqrt{10^{-\text{RetLoss}/10}}\ \texttt{// N}$$

```
0.00177828 Ein
```

High-Power Polarizaion Calculations for Polarization Controller

```
αHighIn (* input high-power orientation *)
ϕHighIn (* input high-power ellipticity *)
```

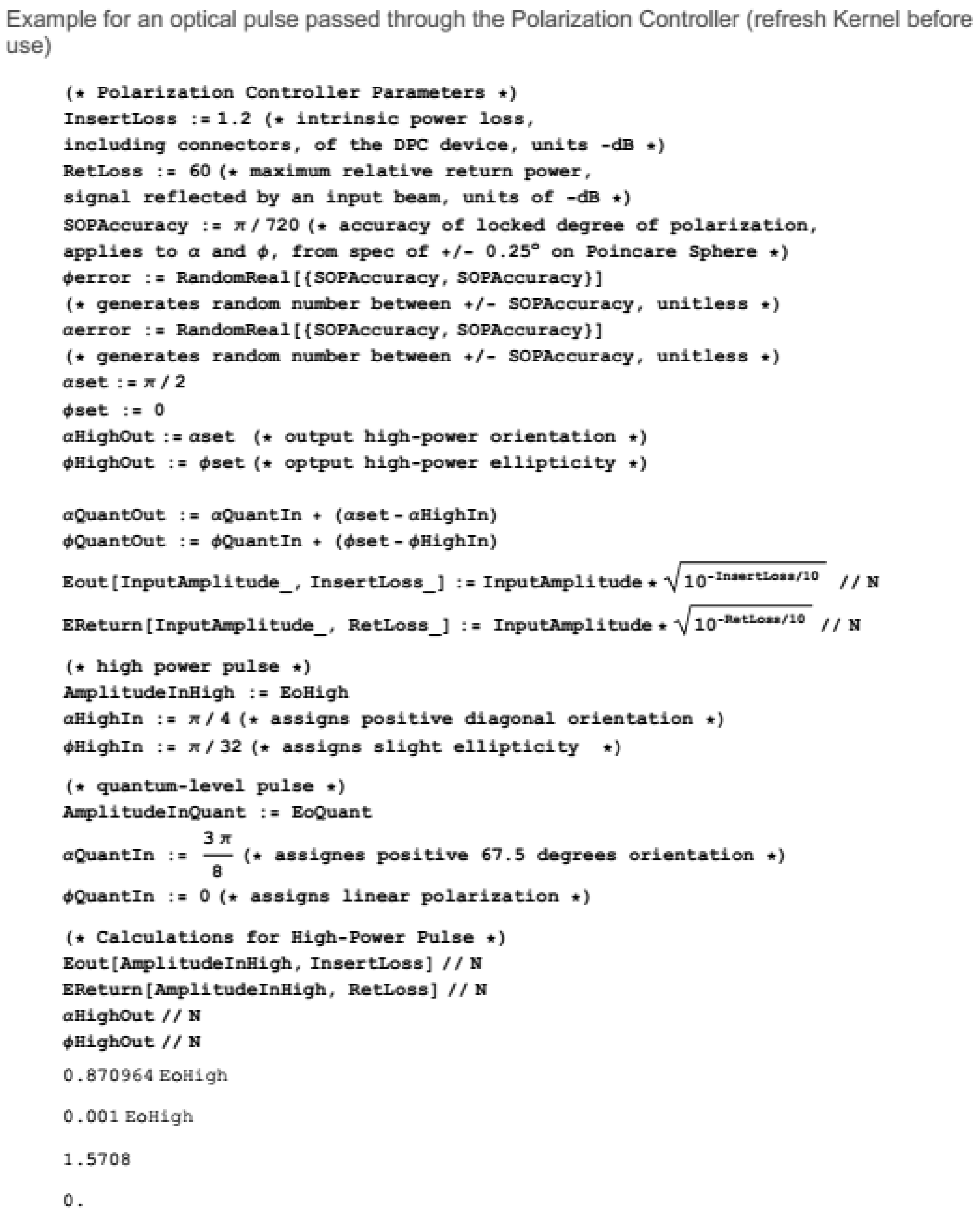
The advantage of the DPC (integrated polarization controller and digital signal processor) is that is can accomodate any input polarization and ellipticity and, with very low loss, lock the output polarization of a beam (pulses) of sufficient power to a pre-selected state.

```
αHighOut := αset (* output high-power orientation;
αset is the orientation setting of the polarimeter *)
ϕHighOut := ϕset (* optput high-power ellipticity;
ϕset is the ellipticity setting of the polarimeter *)
```

Quantum-level Polarization Calculations for Polarization Controller

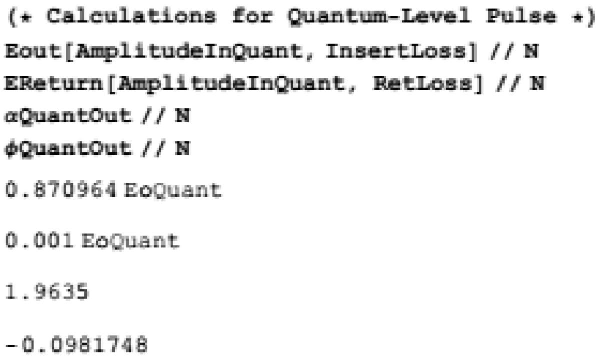
The polarization controller state is set and used to alter the high-power frame signals. This setting, (orientation and ellipticity rotations) used to alter the high-power signals, will also be experienced by the quantum-level pulses. Although factors such as birefringence and wavelength-dependent polarization changes can alter how the polarization controller rotates the quantum-level signal, we can use the following as a first approximation;

```
αQauntOut := αQauntIn + (αset - αHighIn)
ϕQuantOut := ϕQuantIn + (ϕset - ϕHighIn)
```

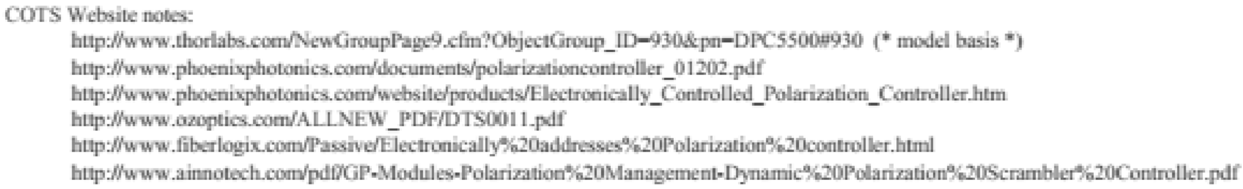


Example for an optical pulse passed through the Polarization Controller (refresh Kernel before use)

```
(* Polarization Controller Parameters *)
InsertLoss := 1.2 (* intrinsic power loss,
including connectors, of the DPC device, units -dB *)
RetLoss := 60 (* maximum relative return power,
signal reflected by an input beam, units of -dB *)
SOPAccuracy := π / 720 (* accuracy of locked degree of polarization,
applies to α and ϕ, from spec of +/- 0.25° on Poincare Sphere *)
ϕerror := RandomReal[{SOPAccuracy, SOPAccuracy}]
(* generates random number between +/- SOPAccuracy, unitless *)
αerror := RandomReal[{SOPAccuracy, SOPAccuracy}]
(* generates random number between +/- SOPAccuracy, unitless *)
αset := π / 2
ϕset := 0
αHighOut := αset  (* output high-power orientation *)
ϕHighOut := ϕset (* optput high-power ellipticity *)

αQuantOut := αQuantIn + (αset - αHighIn)
ϕQuantOut := ϕQuantIn + (ϕset - ϕHighIn)
Eout[InputAmplitude_, InsertLoss_] := InputAmplitude * √(10^(-InsertLoss/10))  // N
EReturn[InputAmplitude_, RetLoss_] := InputAmplitude * √(10^(-RetLoss/10))  // N
(* high power pulse *)
AmplitudeInHigh := EoHigh
αHighIn := π / 4 (* assigns positive diagonal orientation *)
ϕHighIn := π / 32 (* assigns slight ellipticity  *)
(* quantum-level pulse *)
AmplitudeInQuant := EoQuant
αQuantIn := 3 π / 8 (* assignes positive 67.5 degrees orientation *)
ϕQuantIn := 0 (* assigns linear polarization *)
(* Calculations for High-Power Pulse *)
Eout[AmplitudeInHigh, InsertLoss] // N
EReturn[AmplitudeInHigh, RetLoss] // N
αHighOut // N
ϕHighOut // N
0.870964 EoHigh

0.001 EoHigh

1.5708

0.
```



```
(* Calculations for Quantum-Level Pulse *)
Eout[AmplitudeInQuant, InsertLoss] // N
EReturn[AmplitudeInQuant, RetLoss] // N
αQuantOut // N
ϕQuantOut // N
0.870964 EoQuant

0.001 EoQuant

1.9635

-0.0981748
```



```
COTS Website notes:
        http://www.thorlabs.com/NewGroupPage9.cfm?ObjectGroup_ID=930&pn=DPC5500#930  (* model basis *)
        http://www.phoenixphotonics.com/documents/polarizationcontroller_01202.pdf
        http://www.phoenixphotonics.com/website/products/Electronically_Controlled_Polarization_Controller.htm
        http://www.ozoptics.com/ALLNEW_PDF/DTS0011.pdf
        http://www.fiberlogix.com/Passive/Electronically%20addresses%20Polarization%20controller.html
        http://www.ainnotech.com/pdf/GP-Modules-Polarization%20Management-Dynamic%20Polarization%20Scrambler%20Controller.pdf
```

O.9 Component Use Case

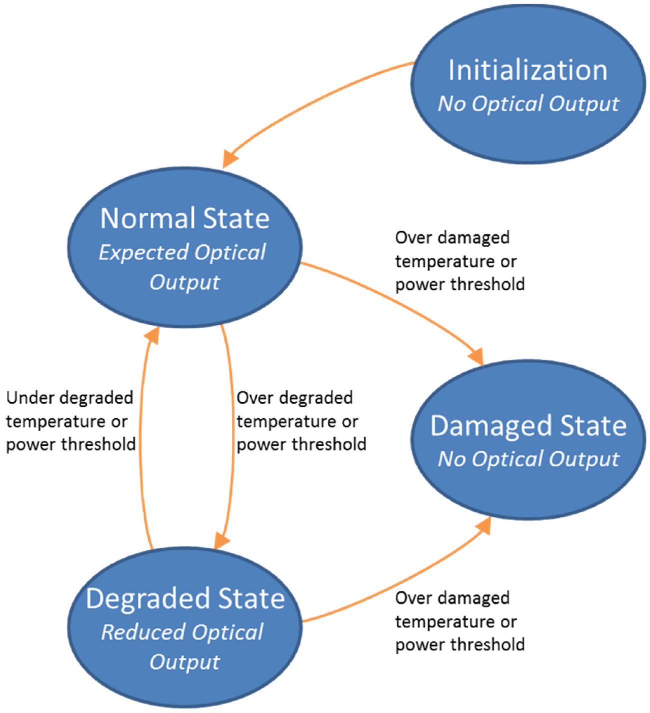
O.9.1 Respond to an Optical Packet in the Polarization Controller

Optical packet arrives at the polarization controller. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the polarization controller. Records overpower condition, if applicable. Remove the optical packet from the queue and checks if it is a bright pulse. If a bright pulse, update packet rotation values. Calculate the attenuated optical output signal and rotate pulse based on the input signal and the current component state. Propagate the attenuated and rotated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
  - Respond to a control message
  - React to an environmental message

- Assumptions

- Component has completed initialization sequence at least once
- Reflections are not affected by component state
- Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 145. Component states.

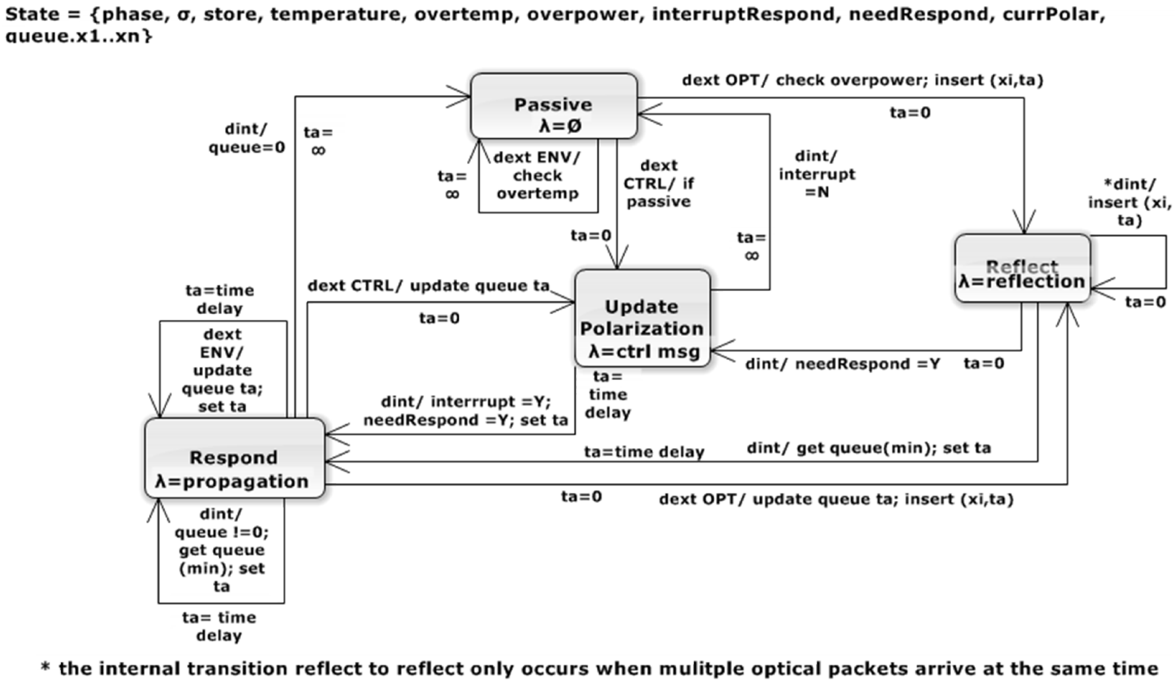

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, currPolar, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dint/ queue=0
ta= ∞
dext ENV/ check overtemp
ta= ∞
dext CTRL/ if passive
ta=0
dint/ interrupt =N
ta= ∞
*dint/ insert (xi, ta)
Reflect
λ=reflection
ta=0
ta=time delay
dext CTRL/ update queue ta
ta=0
Update Polarization
λ=ctrl msg
dext ENV/ update queue ta; set ta
dint/ interrrupt =Y; needRespond =Y; set ta
ta= time delay
dint/ needRespond =Y
ta=0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
ta=0
dext OPT/ update queue ta; insert (xi,ta)
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 146. Polarization controller phase transition diagram.

O.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Polarization rotation values changed upon receipt of a bright pulse
- Optical packet properly attenuated and rotated to the limit of accuracy
- Optical packet propagated out the correct port at the correct time

O.9.3 Respond to an Environmental Packet in the Polarization Controller

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
  - None

O.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly

- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

O.9.5 Respond to a Control Message in the Polarization Controller

Control Message arrives at the component. Component decodes message properly. Records change in condition or state, if applicable. Change component function if in degraded or damaged state or by change in component condition, if necessary.

- Assumptions
    - Component has completed initialization sequence at least once

O.9.6 Respond to Control Message End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

O.10 Polarization Controller Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change

in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *Polarization Controller Test Cases.*

		Inject Ports					Running Totals		
							opt	**env**	**ctrl**
Phase	**Case**	**Opt1**	**Opt2**	**Ctrl**	**Env**	**Notes**	#	**#**	**#**
Passive	1	1	0	0	0	single	1	0	0
	2	0	1	0	0	single	2	0	0
	3	0	0	1	0	single	2	0	1
	4	0	0	0	1	single	2	1	1
	5	1	1	0	0	same time	4	1	1
	6	1	0	1	0	same time	5	1	2
	7	1	1	0	0	differ time	7	1	2
	8	1	0	1	0	differ time	8	1	3
	9	1	1	1	1	same time	10	2	4
	10	1	1	1	1	differ time	12	3	5
	11	0	1	0	1	same time	13	4	5
	12	0	1	0	1	differ time	14	5	5

	13	0	0	1	1	same time	14	6	6
	14	0	0	1	1	differ time	14	7	7
	15	1	0	0	1	same time	15	8	7
	16	1	0	0	1	differ time	16	9	7
	20	2	0	0	0	same time	18	9	7
	21	0	2	0	0	same time	20	9	7
	22	2	1	0	0	same time	23	9	7
	23	2	0	1	0	same time	25	9	8
	24	2	0	0	1	same time	27	10	8
	25	2	0	1	0	differ time	29	10	9
	26	2	1	1	1	same time	32	11	10
	27	2	1	1	1	differ time	35	12	11
	28	0	2	0	1	same time	37	13	11
	29	0	2	0	1	differ time	39	14	11
	30	0	0	1	1	same time	39	15	12
	31	0	0	1	1	differ time	39	16	13
	32	2	0	0	1	same time	41	17	13
	33	2	0	0	1	differ time	43	18	13
totals		27	16	13	18				
Respond	41	2	0	0	0	single	45	18	13
	42	1	1	0	0	single	47	18	13
	43	1	0	1	0	single	48	18	14
	44	1	0	0	1	single	49	19	14
	45	2	1	0	0	same time	52	19	14
	46	2	0	1	0	same time	54	19	15
	47	2	0	0	1	differ	56	20	15

					time			
48	2	0	1	0	differ time	58	20	16
49	2	1	1	1	same time	61	21	17
50	2	1	1	1	differ time	64	22	18
51	1	1	0	1	same time	66	23	18
52	1	1	0	1	differ time	68	24	18
60	3	0	0	0	same time	71	24	18
61	1	2	0	0	same time	74	24	18
62	3	1	0	0	same time	78	24	18
63	3	0	1	0	same time	81	24	19
64	3	0	0	1	same time	84	25	19
65	3	0	1	0	differ time	87	25	20
66	3	1	1	1	same time	91	26	21
67	3	1	1	1	differ time	95	27	22
68	1	2	0	1	same time	98	28	22
69	1	2	0	1	differ time	101	29	22
totals	43	15	9	11				
TC1	1	0	1	2	single	102	31	23
TC2	1	0	1	2	single	103	33	24
TC3	1	0	1	2	single	104	35	25
TC4	1	0	1	2	single	105	37	26
TC5	1	0	1	2	single	106	39	27
TC6	1	0	1	2	single	107	41	28
TC7	1	0	0	2	single	108	43	28
TC8	1	0	0	2	single	109	45	28
totals	8	0	6	16	s			

Notes: 23 - INIT control message sent; OPT1 & Ctrl - same time - Passive:

25 - Set polarization message & bright pulse - OPT1 & Ctrl - differ time - Passive: sent value of 2.1(pol), PI(orient) and 0.7(ellip) for bright pulse

26 - Get Polarization control message sent - OPT1, OPT2, Ctrl & ENV - same time - Passive:

should respond with 1.5707963, weak bright pulse

30 - INIT control message sent - Ctrl & ENV - same time - Passive:

46 - Set polarization control message sent - OPT1 & Ctrl - same time - Passive: sent value of -2

48 - Get Polarization control message sent - OPT1 & Ctrl - differ time - Passive: should respond with -2

63 - INIT control message sent - OPT1 & Ctrl - same time - Respond:

67 - INIT control message sent - OPT1, OPT2, Ctrl & ENV - differ time - Respond:

O.11 References

ThorLabs. (2013). Deterministic polarization controller - DPC5500. Retrieved October 01, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=930

Appendix P - Polarization Modulator (PM)

P.1 Device Description:

The polarization modulator (PM) is an abstract component that represents any number of devices used to electronically change the polarization of the light stream. In practice there are many ways to accomplish this change in polarization, or generate the polarizations, as is done by using four different lasers, each with a different polarization. This research conceptualizes these devices as having some form of polarization material that can be moved. The effect is to change a known polarization to into one of several output polarizations. Unlike a deterministic polarization controller which can automatically determine the input polarization and make changes to produce a single desired output, the PM responds to external commands to set the output polarization to a fixed level and cannot determine the input polarization.

Conceptually, the PM is an in-line bidirectional optical component with two optical ports. Optical signals arriving at one of the ports is attenuated and polarized, then propagated to the other port after a defined propagation delay. The PM is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the PM may become permanently damaged which changes its attenuation characteristics. Similarly, the PM is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the PM may become temporarily degraded or permanently damaged which changes its attenuation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a "normal" operating temperature.

The first step involved with the modeling the PM is to collect and understand the physical, behavioral, and performance characteristics of components that might be used to construct such a device, such as the in-line polarizer and the electronically variable optical attenuator. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the PM. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the PM to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the PM using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the PM. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

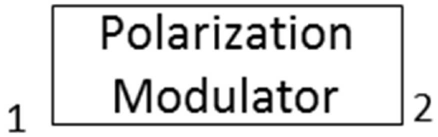


Figure 147. Symbol for the PM in the QKD system architecture.

P.2 PM Conceptual Model

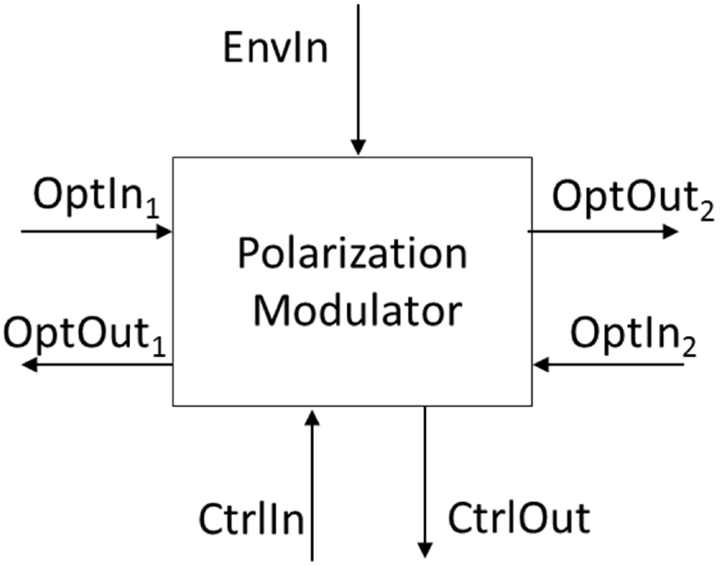


Figure 148. PM conceptual model.

The conceptual model for an PM consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, one environmental input port {EvnIn} and one electrical controller input port and one electrical controller output port {CtrlIn, CtrlOut}. The environmental port allows external sources to communicate changes in the operational environment to the PM. The electrical controller ports allow for control inputs to the controller and responses from the PM to the higher system functions.

In comparison to the PM symbol used in the QKD simulation architecture shown in Figure 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. The electrical control port is not shown for clarity in Figure 2, and is also decomposed in the model into an input port and an output port. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the PM, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The PM must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the PM can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the polarization, attenuation and propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

P.3 Mathematical Model

For a detailed mathematical description of the PM, refer to Section 14.8 which contains the Mathematica worksheet provided by the optical physics SME.

P.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the PM.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.

- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue
- Calculate the reflected power of the signal and send its output with the same port number.
- Retrieve the input optical signal from the queue, and calculate the attenuated output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment
- Update the values of the input optical signal based on the characteristics of the PM, the original values of the input optical signal and the current environment.
- Send the changed output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

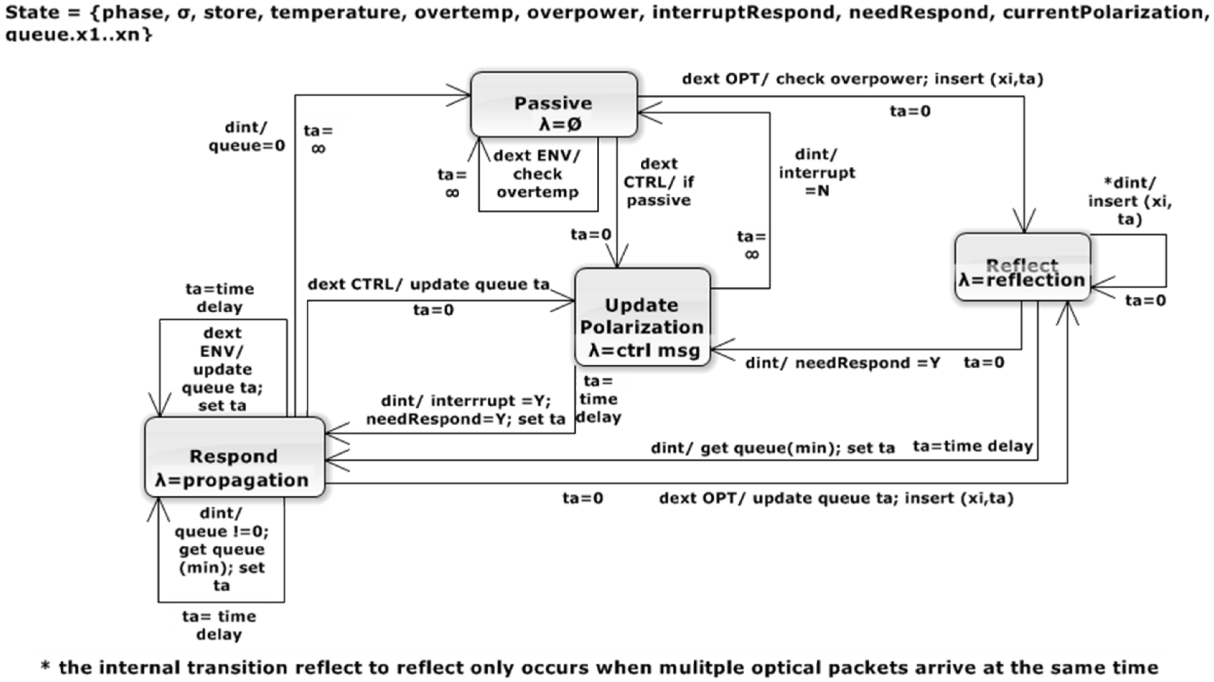
- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

When a control message arrives:

- Change the output polarization per the control message.

P.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the PM in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the PM at the same time.


State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, currentPolarization, queue.x1..xn}

* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 149. PM phase transition diagram.

P.6 Event-Trace Diagram

This section shows various examples of packets entering the PM. The tables list the states the PM proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the PM, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state

- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

P.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 61. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	no env	no ext	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	θ	null	
0	s0	exit	passive	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	θ	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	θ	null	
5	s2	exit	respond	inf	x1	c	n	n	n	n	θ	null	
5	s3	entry	passive	inf	x1	c	n	n	n	n	θ	null	

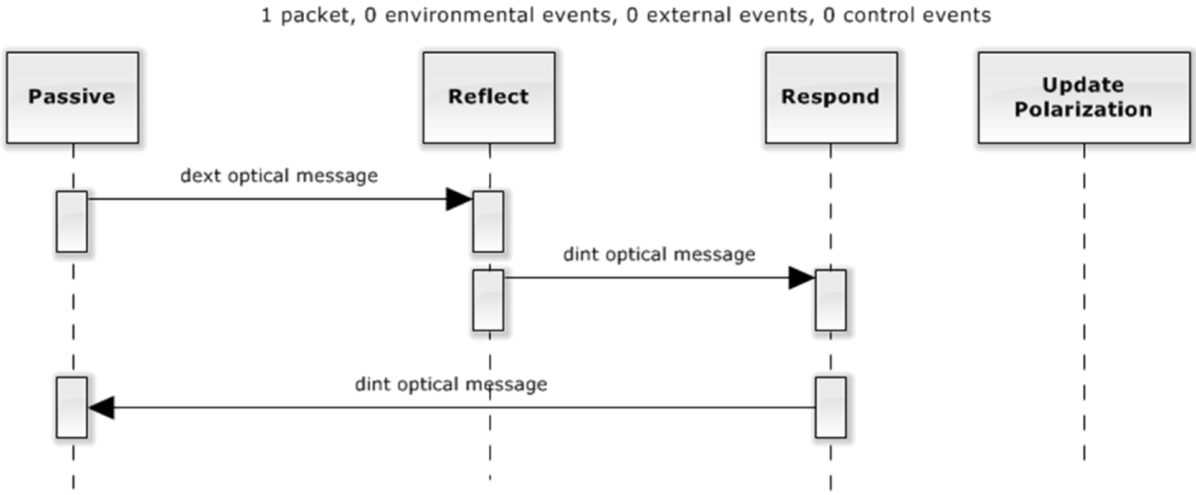


Figure 150. Case I sequence diagram.

P.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 62. *Case II state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1- packet	0 env	1 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	θ	null	
0	s0	exit	passive	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	θ	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	θ	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	θ	(x2,5)	dext at e=2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	θ	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	θ	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	θ	(x2,5)	
5	s4	exit	respond	2	x2	c	n	n	n	n	θ	null	
5	s5	entry	respond	2	x2	c	n	n	n	n	θ	null	
7	s5	exit	respond	inf	x2	c	n	n	n	n	θ	null	
7	s6	entry	passive	inf	x2	c	n	n	n	n	θ	null	

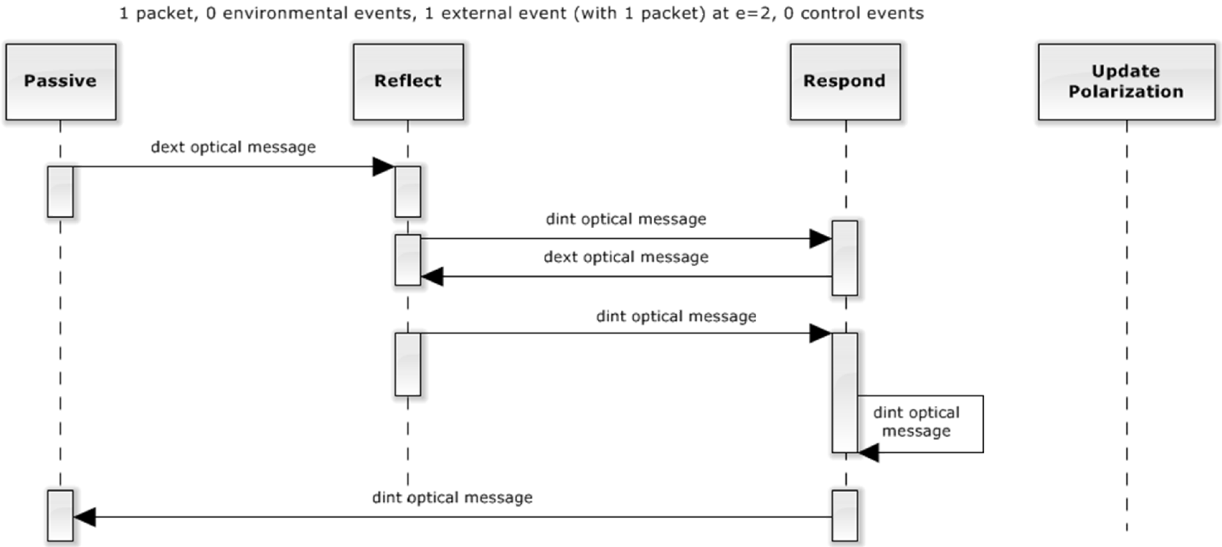


Figure 151. Case II sequence diagram.

P.6.3 *CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3*

Table 63. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (x*i*, *tp*)**	**Notes: assume tp= 5**
	1-packet	0 env	2 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	θ	null	
0	s0	exit	passive	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	θ	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	θ	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	θ	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	θ	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	θ	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	θ	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	n	θ	(x2,4) (x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	n	θ	(x2,4) (x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	n	θ	(x2,4) (x3,5) (x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	n	θ	(x2,4) (x3,5) (x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	n	θ	(x2,4) (x3,5) (x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	n	θ	(x2,4) (x3,5) (x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	n	θ	(x3,2) (x4,2)	
5	s8	entry	respond	2	x2	c	n	n	n	n	θ	(x3,2) (x4,2)	
7	s8	exit	respond	1	x3	c	n	n	n	n	θ	(x4,0)	
7	s9	entry	respond	1	x3	c	n	n	n	n	θ	(x4,0)	
8	s9	exit	respond	0	x4	c	n	n	n	n	θ	null	
8	s10	entry	respond	0	x4	c	n	n	n	n	θ	null	
8	s10	exit	respond	inf	x4	c	n	n	n	n	θ	null	
8	s11	entry	passive	inf	x4	c	n	n	n	n	θ	null	

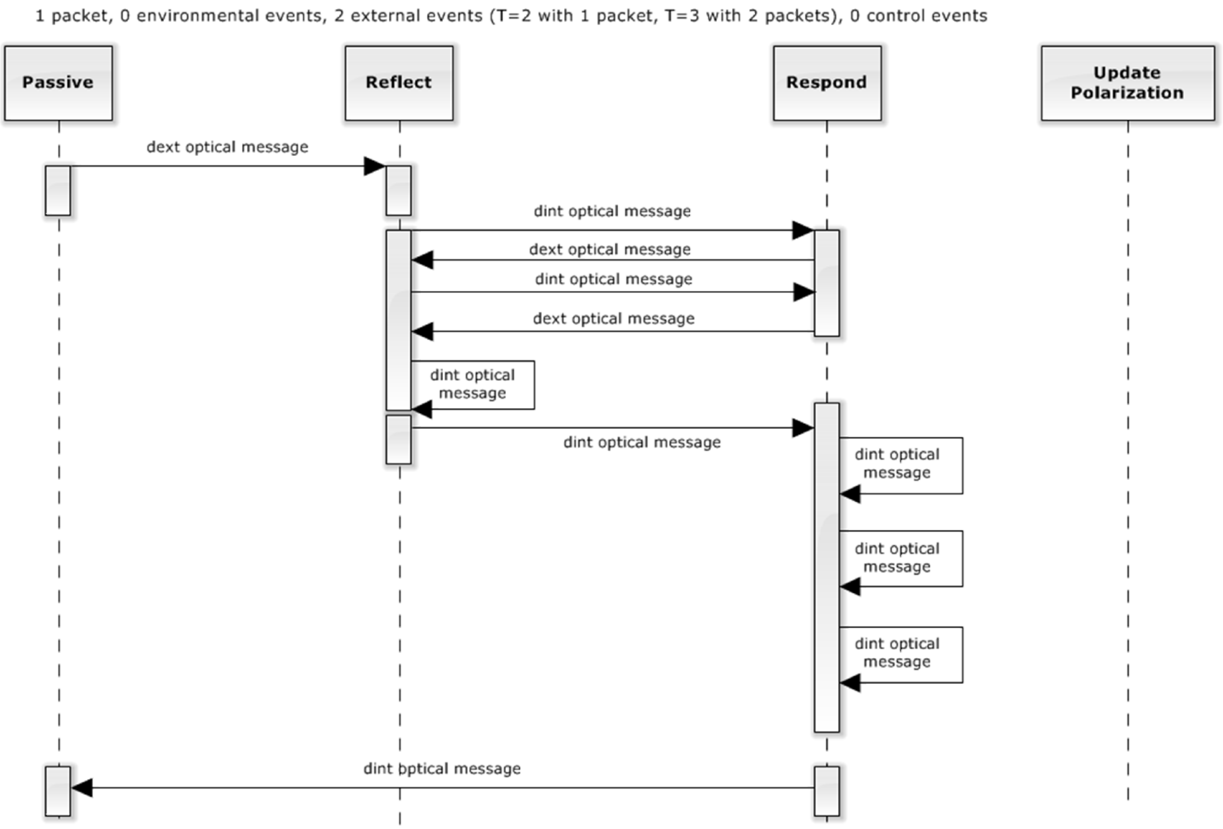


Figure 152. Case III sequence diagram.

P.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 64. *Case IV state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	1 env	0 ext	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	*θ*	null	
0	s0	exit	passive	0	null	c	n	n	n	n	*θ*	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	*θ*	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	*θ*	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	*θ*	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	y	n	*θ*	null	update temp
3	s3	entry	respond	2	x1	c	y	n	y	n	*θ*	null	
5	s3	exit	respond	inf	x1	c2	y	n	n	n	*θ*	null	

5	s4	entry	passive	inf	x1	c2	y	n	n	n	θ	null	

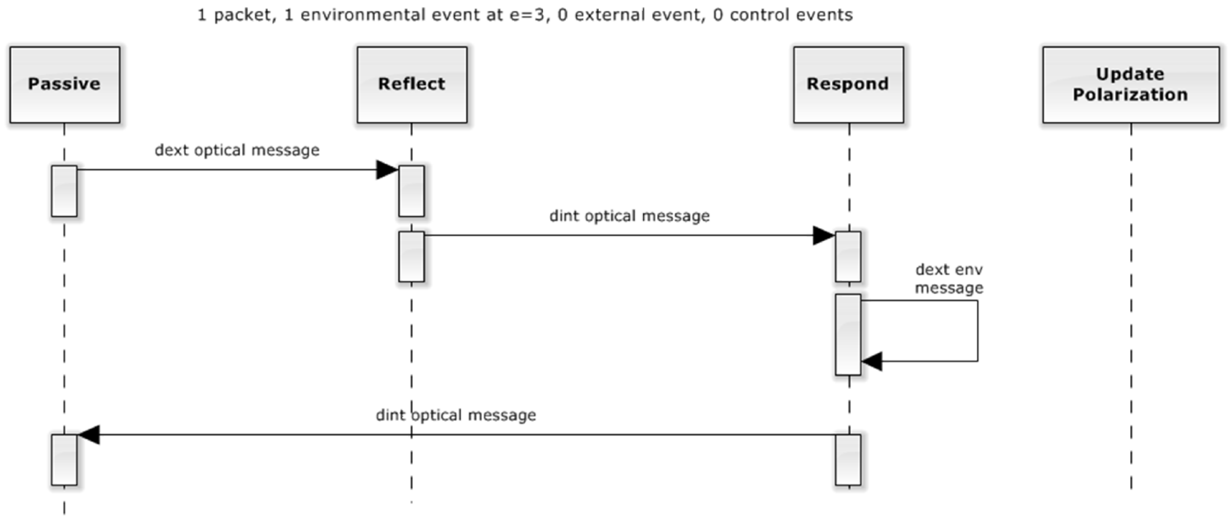


Figure 153. Case IV sequence diagram.

P.6.5 CASE V: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Control Packet Arriving at Time 3

Table 65. *Case V state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**current polar**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1 opt	1 env	0 opt	1 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	θ	null	
0	s0	exit	passive	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	θ	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	θ	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	θ	(x1,5)	CTRL arrives e=3
3	s2	exit	respond	0	x1	c	n	n	y	n	θ	(x1,2)	
3	s3	entry	update detector	0	x1	c	n	n	y	n	θ	(x1,2)	
3	s3	exit	update detector	2	x1	c	n	n	y	n	θ	(x1,2)	
3	s4	entry	respond	2	x1	c	n	n	y	n	θ	(x1,2)	
5	s4	exit	respond	0	x1	c	n	n	n	n	θ	null	
5	s6	entry	passive	inf	x1	c	n	n	n	n	θ	null	

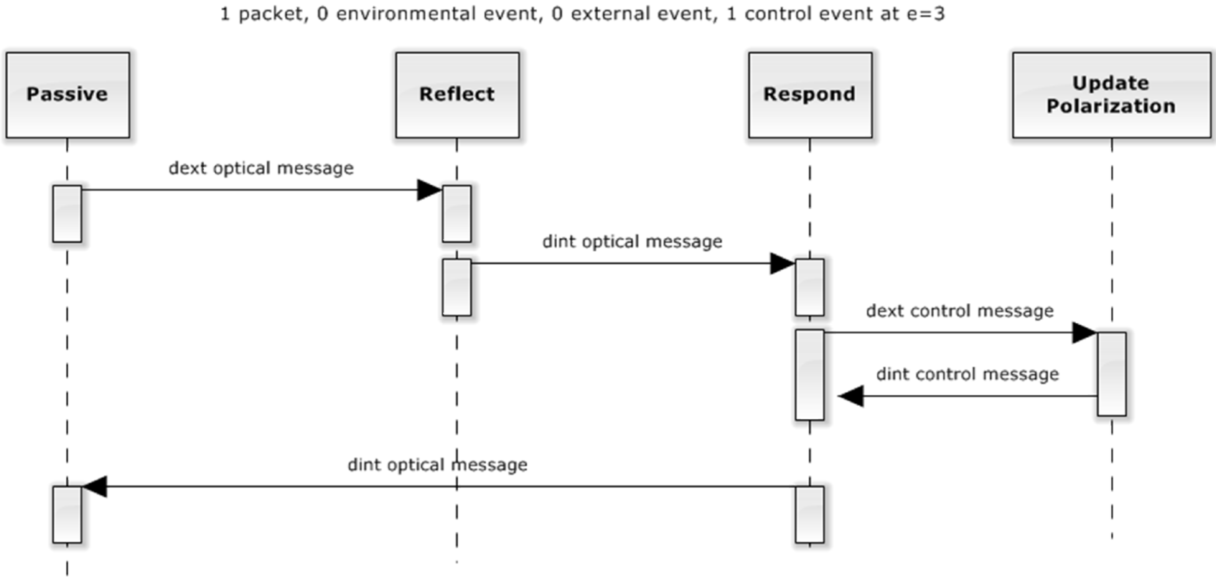


Figure 154. Case V sequence diagram.

P.7 PM Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- Assume that only one control packet will arrive at any given time, due to the small time scales involved and the length of time necessary for attenuation changes.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”,”interruptRespond”, “needRespond”, “currentPolarization”,queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values

“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the PM we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS$_{PM}$* = (*X_M*, *Y_M*, *S*, *δ_{ext}*, *δ_{int}*, *δ_{con}*, *λ*, *ta*)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “reflect”, “respond”, “update detector”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
needRespond= flag variable set when both Reflect and UpdateDetector respond to inputs
currentPolarization = current polarization of the PM
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (p,v)
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port

reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
ctrlOutput = variable that stores the output control message response
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
attenuationMin = minimum selectable attenuation
attenuationMax = maximum selectable attenuation
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
ctrlMsg() = method that generates a response message to received control messages
outputMsg() = method that generates the response message to received optical packets
insert_event_q() = method that inserts the current (x_i, time $delay_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
changePolarization() = method that changes current polarization of the PM
calcAttenPolar() = method that calculates the optical output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr, currentPolarization*)
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue

$q.v_{min}$ = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“OptIn$_1$”, “OptIn$_2$”, “EnvIn”, “CtrlIn”} with
X_M = {(“OptIn$_1$”, V_{opt}), (“OptIn$_2$”, V_{opt}), (“EnvIn”, V_{env}), (“CtrlIn”, V_{ctrl})} is the set of input ports and values.

OutPorts = {“OptOut$_1$”, “OptOut$_2$”, “CtrlOut”} with
Y_M = {(“OptOut$_1$”, $Y_{OptOut1}$), (“OptOut$_2$”, $Y_{OptOut2}$), (“CtrlOut”, $Y_{CtrlOut}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {phase, σ, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, needRespond, currentPolarization, queue*} = {{“passive”, “reflect”, “respond”, “update polarization”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x {“Y”,”N”} x *V* x *V*}

External Transition Function:

δ_{ext}(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond , currentPolarization, queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =

```
(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                  currentPolarization, queue.x1..xn)
  if phase = “passive” and p ∈ {“OptIn1”, “OptIn2”}
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower =  “Y”
      insert_event_q(current)
      remove_event_m(current)
    queue.current = queue.first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                  currentPolarization, queue.x1..xn)
  if phase = “respond” and p ∈ {“OptIn1”, “OptIn2”}
    update_delay(queue)
    for messagebag != null
     current = messagebag_first()
```

```
      if current.value.power > damaged.power
        overpower =  “Y”
      insert_event_q(current)
      remove_event_m(current)
    queue.current = queue_need_reflected()
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond= “Y”
   timeLeftRespond = timeLeftRespond - e

(“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                currentPolarization, queue.x1..xn)
   if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                    needRespond,  currentPolarization, queue.x1..xn)
   if phase = “respond” and p = “EnvIn”
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”
     time.delay = timeLeftRespond

(“update polarization”, 0, store, temperature, overtemp, overpower, interruptRespond,
                                    needRespond, currentPolarization, queue.x1..xn)
   if phase = “passive” and p =“CtrlIn”
     ctrlOutput = ctrlMsg(store)
     currentPolarization = store.value.polarization

(“update polarization”, 0, store, temperature, overtemp, overpower, interruptRespond,
                                    needRespond, currentPolarization, queue.x1..xn)

   if phase = “respond” and p = “CtrlIn”
     update_delay(queue)
     ctrlOutput = ctrlMsg(store)
     interruptRespond= “Y”
     currentPolarization = store.value.polarization

(phase, σ − e, ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                currentPolarization, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                    currentPolarization, queue)=
("reflect", 0, temperature, overtemp, overpower, interruptRespond, needRespond,
                                              currentPolarization, queue.x1..xn))
  if phase = "reflect" and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                   needRespond, currentPolarization, queue.x1..xn)
  if phase = "reflect" and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = "N"
       current = queue_min()
       time.delay = current.time.delay
       if InPort = "OptIn$_1$"
         outputPulse = calcAttenPolar(store.v, temperature, overtemp, peakpwr, overpwr)
         outputPort = "OptOut$_2$"
      if InPort = "OptIn$_2$"
        outputPulse = calcAttenPolar(store.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "OptOut$_1$"
      timeLeftRespond = propagation delay
   else
     time.delay = timeLeftRespond

 ("update polarization", 0, store, temperature, overtemp, overpower, interruptRespond,
                                   needRespond, currentPolarization, queue.x1..xn)
   if phase = "reflect" and needRespond = "Y"
     ctrlOutput = ctrlMsg(store)

 ("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                   needRespond, currentPolarization, queue.x1..xn)
   if phase = "respond" and size > 0
     update_delay(queue)
     size= queue_size()
     current = queue_min()
     time.delay = current.time.delay
     if InPort = "OptIn$_1$"
       outputPulse = calcAttenPolar(store.v, temperature, overtemp, peakpwr, overpwr)
       outputPort = "OptOut$_2$"
     if InPort = "OptIn$_2$"
       outputPulse = calcAttenPolar(store.v, temperature, overtemp, peakpwr, overpwr)
    outputPort = “OptOut1”
   interruptRespond= “N”

 (“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                       currentPolarization, queue.x1..xn)
   if phase = “respond” and size  = 0
    size= queue_size()

 (“passive”, ∞, store, temperature, overtemp, overpower, overpower, interruptRespond,
                                          needRespond, currentPolarization, queue.x1..xn)
   if phase = “update polarization” and interruptRespond =  “N”

 (“respond”, time.delay, store, temperature, overtemp, overpower, overpower, interruptRespond,
                                          needRespond, currentPolarization, queue.x1..xn)
   if phase = “update polarization” and interruptRespond = “Y”
    time.delay = timeLeftRespond

 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                          needRespond, currentPolarization, queue.x1..xn)
   if phase = “update polarization” and interruptRespond = “N” and needRespond = “Y”
    current = queue_min()
    time.delay = current.time.delay
    if InPort = “OptIn1”
      outputPulse = calcAttenPolar(store.v, temperature, overtemp, peakpwr, overpwr)
      outputPort = “OptOut2”
    if InPort = “OptIn2”
      outputPulse = calcAttenPolar(store.v, temperature, overtemp, peakpwr, overpwr)
      outputPort = “OptOut1”
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

```
λ(phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                          currentPolarization, queue) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (outputPort, outputPulse)
    if phase = “respond”

  (“CtrlOut”, ctrlOutput)
    if phase = “update polarization”

  ∅ (null output)
```

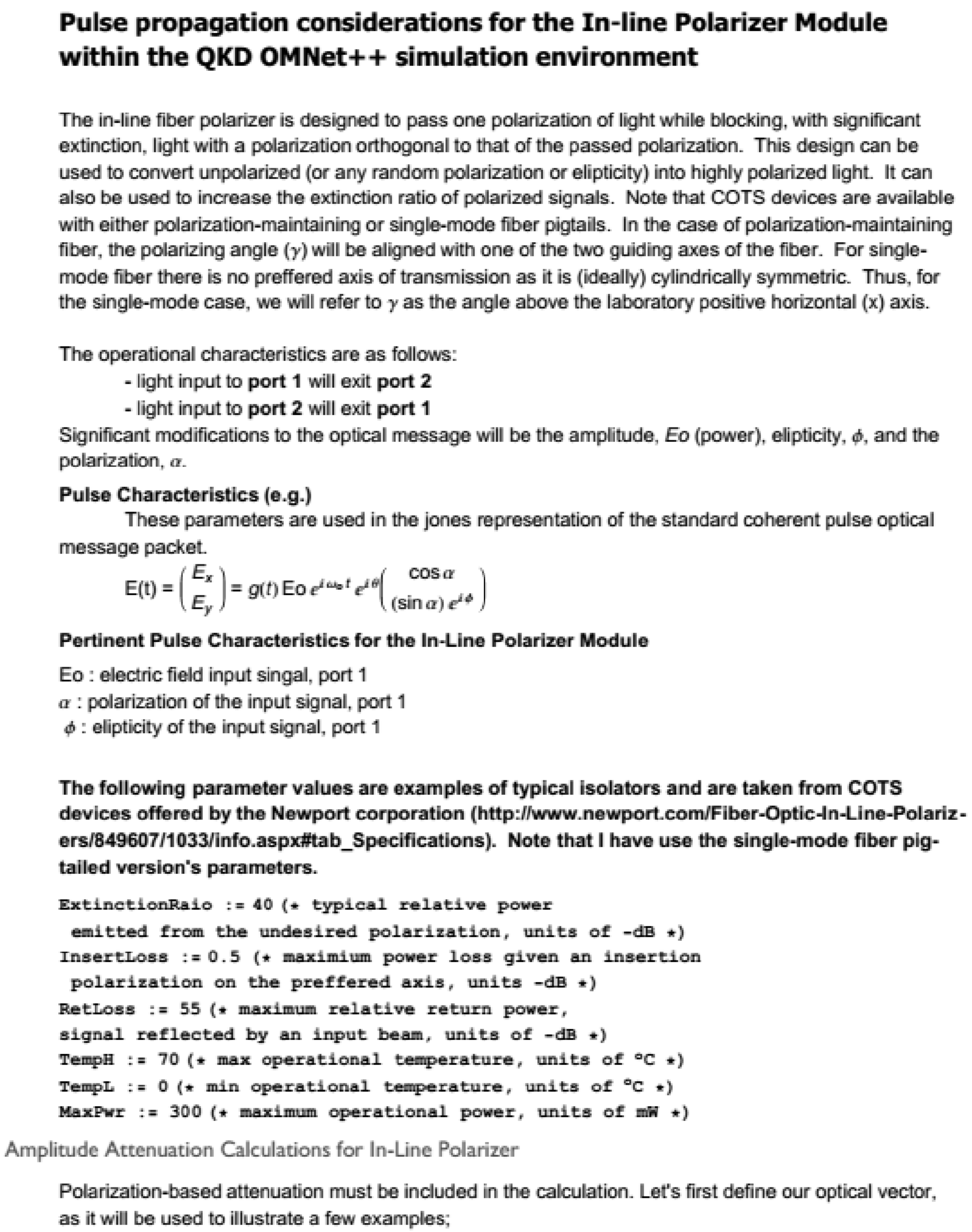
otherwise;

Time advance Function:

ta(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond, currentPolarization, queue*) = *σ*;

P.8 Mathematical Model

Below is the math model for the in-line polarizer. The polarization modulator works in much the same way with the addition of a control port to allow the output polarization angle (α) to be changed during operation.



Pulse propagation considerations for the In-line Polarizer Module within the QKD OMNet++ simulation environment

The in-line fiber polarizer is designed to pass one polarization of light while blocking, with significant extinction, light with a polarization orthogonal to that of the passed polarization. This design can be used to convert unpolarized (or any random polarization or elipticity) into highly polarized light. It can also be used to increase the extinction ratio of polarized signals. Note that COTS devices are available with either polarization-maintaining or single-mode fiber pigtails. In the case of polarization-maintaining fiber, the polarizing angle (γ) will be aligned with one of the two guiding axes of the fiber. For single-mode fiber there is no preffered axis of transmission as it is (ideally) cylindrically symmetric. Thus, for the single-mode case, we will refer to γ as the angle above the laboratory positive horizontal (x) axis.

The operational characteristics are as follows:

- light input to **port 1** will exit **port 2**
- light input to **port 2** will exit **port 1**

Significant modifications to the optical message will be the amplitude, *Eo* (power), elipticity, ϕ, and the polarization, α.

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\, e^{i\,\omega_o\,t}\, e^{i\,\theta} \begin{pmatrix} \cos\alpha \\ (\sin\alpha)\, e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the In-Line Polarizer Module

Eo : electric field input singal, port 1

α : polarization of the input signal, port 1

ϕ : elipticity of the input signal, port 1

The following parameter values are examples of typical isolators and are taken from COTS devices offered by the Newport corporation (http://www.newport.com/Fiber-Optic-In-Line-Polarizers/849607/1033/info.aspx#tab_Specifications). Note that I have use the single-mode fiber pigtailed version's parameters.

```
ExtinctionRaio := 40 (* typical relative power
 emitted from the undesired polarization, units of -dB *)
InsertLoss := 0.5 (* maximium power loss given an insertion
 polarization on the preffered axis, units -dB *)
RetLoss := 55 (* maximum relative return power,
signal reflected by an input beam, units of -dB *)
TempH := 70 (* max operational temperature, units of °C *)
TempL := 0 (* min operational temperature, units of °C *)
MaxPwr := 300 (* maximum operational power, units of mW *)
```

Amplitude Attenuation Calculations for In-Line Polarizer

Polarization-based attenuation must be included in the calculation. Let's first define our optical vector, as it will be used to illustrate a few examples;

$$\mathtt{OptVect := A\, e^{i\, Re[\omega o\, t]}} \begin{pmatrix} \mathtt{Cos[\alpha]} \\ \mathtt{Sin[\alpha]\, e^{i\,\phi}} \end{pmatrix}$$

Here, *A* is the input amplitude, α is the polarization, and ϕ is the ellipticity.

For the case of single-mode fiber pigtails we use the horizontal lab frame as γ=0 (vertical as γ=π/2), where γ can be any value [0 : π/2). For the case of polarization-maintaining fiber pigtails, γ will have the values of only 0 or π/2. In either case, the tranformation matrix is,

$$\mathtt{Polarizer[\gamma\_] :=} \begin{pmatrix} \mathtt{(Cos[\gamma])^2} & \mathtt{(Cos[\gamma] * Sin[\gamma])} \\ \mathtt{(Cos[\gamma] * Sin[\gamma])} & \mathtt{(Sin[\gamma])^2} \end{pmatrix}$$

To calculate the output amplitude we first want to account for the insertion loss

$$\mathtt{Etemp[Amplitude\_, InsertionLoss\_] := Amplitude * \sqrt{10^{-InsertionLoss/10}}}$$

Here we use the aforementioned insertion loss of -0.5 dB and the input amplitude Eo

```
EAfterInsertLoss = Etemp[A, InsertLoss]
0.944061 A
```

We now have to calculate the effective amplitude of the electric field after the light has passed through the polarizer. This is accomplished by taking the norm of the vector produced by operating the polarization matrix on the optical vector;

```
Eout[γ_] = Norm[Polarizer[γ].OptVect] // FullSimplify
```

$$\mathtt{Abs\left[Cos[\alpha]\ Cos[\gamma] + e^{i\,\phi}\ Sin[\alpha]\ Sin[\gamma]\right] \sqrt{A\ Conjugate[A]\ Cosh[2\ Im[\gamma]]}}$$

If we wish to flag the attenuator to include **undesired return (reflected)** messages, the following operations would hold true,

$$\mathtt{Eout[Ein\_, RetLoss\_] := Ein * \sqrt{10^{-RetLoss/10}}}$$

Examples For Amplitude Attenuation in the In-Line Polarizer

As an example, let's take the polarizer and input polarization to be oriented in the same, positive diagonal (45^o) with no elipticity;

$$\mathtt{Eout\left[\frac{\pi}{4}\right]\ /.\ \alpha \to \frac{\pi}{4}\ /.\ \phi \to 0\ //\ FullSimplify}$$

```
Abs[A]
```

As it should be, the only loss in the first example is due to insertion loss, which is not included here for illustrative purposes. To include insertion loss, simply define the amplitude *A* to be *EAfterInsertLoss*, i.e.,

```
Eout[π/4] /. α → π/4 /. ϕ → 0 /. A → EAfterInsertLoss // FullSimplify
0.944061 Abs[A]
```

As a second example, again excluding insertion loss, let' s keep the polarizer and input polarization at the same angle, but make the light circular ($\alpha = \pi/4$, $\phi = \pi/2$). Insertion loss is not included hereafter for the sake of clarity;

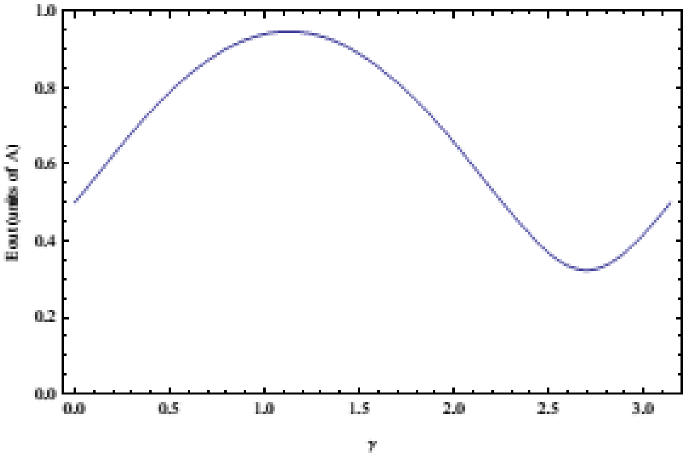
```
Eout[π/4] /. α → π/4 /. ϕ → π/2 // FullSimplify // N
0.707107 Abs[A]
```

Because the polarization "rotates", a power factor of 1/2 will be lost ($1/\sqrt{2}$ in the amplitude). Because the light is circular, the orientation of the polarizer won't effect the output amplitude, even if perpendicular to the initial input "polarization";

```
Eout[3 π/4] /. α → π/4 /. ϕ → π/2 // FullSimplify // N
0.707107 Abs[A]
```

The same doesn't hold true for elliptically polarized light. In the plot below, I change α to $\frac{\pi}{3}$, making the light slightly elliptical. I vary the polarizer angle γ from 0 to π;

```
Plot[Eout[γ] /. α → π/3 /. ϕ → π/4 /. A → 1, {γ, 0, π}, Frame → True,
  PlotRange → {0, 1}, FrameLabel → {"γ", "Eout (units of A)"}]
```



This plot is fun to play with. Note what happens when you have circular light (α=π/4, ϕ=π/2), linear light (α = any angle, ϕ = 0) or any elliptical light (vary α or ϕ).

As a final example, let's put the polarizer angle at horizontal (γ = 0), but change the input polarization to vertical, with no ellipticity

```
Eout[0] /. α → π/2 /. ϕ → 0 // FullSimplify
0
```

This is idealized, as the devices typically call for an extinction ratios of -30 to -40 dB.

Polarizaion Calculations for In-Line Polarizer

In all cases, the light exiting the polarizer will be nearly pure (within the tolerances set by the extinction ratio) in the axis, γ, of the polarizer. Thus, for all cases,

$$\texttt{OutputPol} := \gamma$$

Final Example for an optical pulse passed through the In-Line Polarizer (refresh Kernel before use)

$$\texttt{InsertLoss} := 0.5$$

$$\texttt{OptVect}[\alpha\_, \phi\_] := A\, e^{i\,\mathrm{Re}[\omega o\, t]} \begin{pmatrix} \mathrm{Cos}[\alpha] \\ \mathrm{Sin}[\alpha]\, e^{i\phi} \end{pmatrix}$$

$$\texttt{Polarizer}[\gamma\_] := \begin{pmatrix} (\mathrm{Cos}[\gamma])^2 & (\mathrm{Cos}[\gamma] * \mathrm{Sin}[\gamma]) \\ (\mathrm{Cos}[\gamma] * \mathrm{Sin}[\gamma]) & (\mathrm{Sin}[\gamma])^2 \end{pmatrix}$$

$$\texttt{Eout}[\gamma\_, \alpha\_, \phi\_] = \sqrt{10^{-\texttt{InsertLoss}/10}} * \texttt{Norm}[\texttt{Polarizer}[\gamma].\texttt{OptVect}[\alpha, \phi]]\ //\ \texttt{FullSimplify}$$

$$0.944061\, \mathrm{Abs}\left[\mathrm{Cos}[\alpha]\, \mathrm{Cos}[\gamma] + e^{i\phi}\, \mathrm{Sin}[\alpha]\, \mathrm{Sin}[\gamma]\right] \sqrt{A\, \mathrm{Conjugate}[A]\, \mathrm{Cosh}[2\, \mathrm{Im}[\gamma]]}$$

In the example below, I have assigned the polarizer angle to be just above horizontal ($\gamma = \frac{\pi}{16}$) with the light polarized at positive diagonal ($\alpha = \pi/4$) with a slight ellipticity ($\phi = \pi/32$)

$$\texttt{OutputAmplitude} = \texttt{Eout}\left[\frac{\pi}{16}, \frac{\pi}{4}, \frac{\pi}{32}\right]\ //\ \texttt{FullSimplify}$$

$$0.784435\, \mathrm{Abs}[A]$$

$$\texttt{OutputPolariztion} = \frac{\pi}{16}$$

$$\frac{\pi}{16}$$

$$\texttt{OutputEllipticity} = 0$$

$$0$$

The form of the output optical pulse given the above parameters would be,

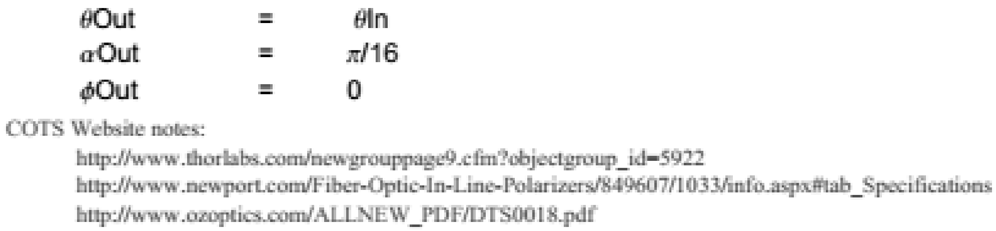
$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = \mathrm{OutpuAmplitude}\, e^{i\omega_o t}\, e^{i\theta} \begin{pmatrix} \mathrm{Cos\,(OuputPolarization)} \\ \mathrm{Sin\,(OutputPolarization)}\, e^{i * \mathrm{OutputEllipticity}} \end{pmatrix}$$

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = 0.784435 ^* A ^* (\, e^{i\,\omega_o\, t}\, e^{i\,\theta}) \begin{pmatrix} \mathrm{Cos}[\pi/16] \\ \mathrm{Sin}[\pi/16]\, e^{i*0} \end{pmatrix}$$

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = 0.784435 ^* A ^* (\, e^{i\,\omega_o\, t}\, e^{i\,\theta}) \begin{pmatrix} 0.980785 \\ 0.19509 \end{pmatrix}$$

in other words,

AmplitudeOut = 0.784435 * AmplitudeIn
ωoOut = ωoIn



θOut	=	θIn
αOut	=	$\pi/16$
ϕOut	=	0

COTS Website notes:

http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=5922
http://www.newport.com/Fiber-Optic-In-Line-Polarizers/849607/1033/info.aspx#tab_Specifications
http://www.ozoptics.com/ALLNEW_PDF/DTS0018.pdf

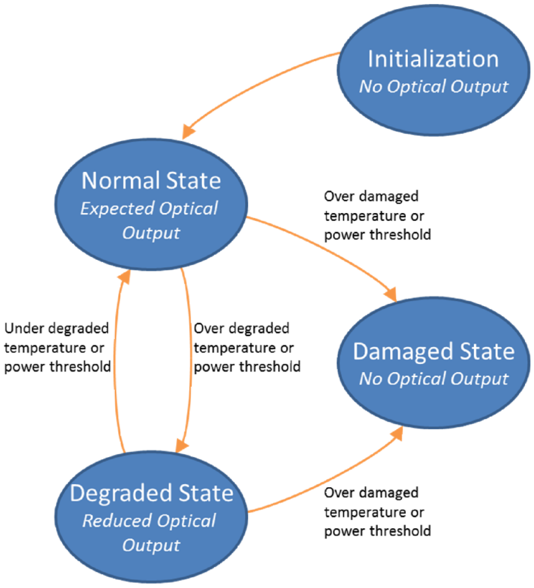
P.9 Component Use Case

P.9.1 Respond to an Optical Packet in the Polarization Modulator (PM)

Optical packet arrives at the polarization modulator. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the polarization modulator. Records overpower condition, if applicable. Remove the optical packet from the queue and change its polarization based on current component state. Calculate the attenuated optical output signal based on the input signal and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
  - Respond to a control message
  - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 155. Component states.

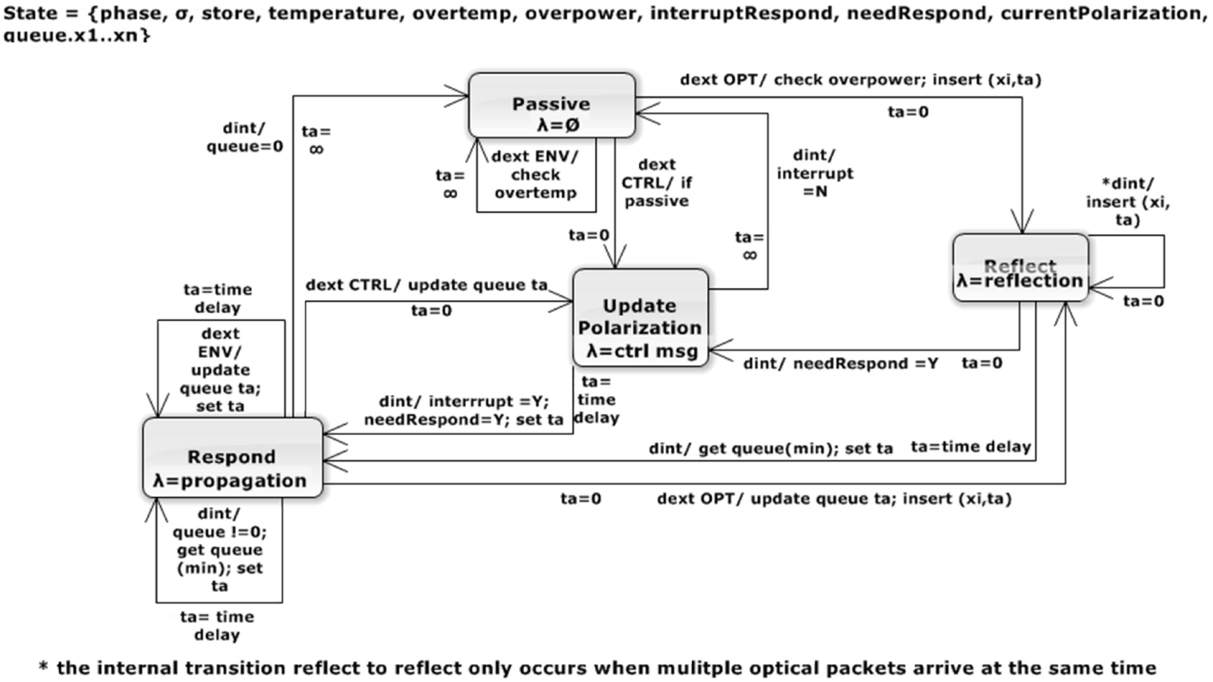

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, currentPolarization, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dint/ queue=0
ta= ∞
ta= ∞
dext ENV/ check overtemp
dext CTRL/ if passive
dint/ interrupt =N
*dint/ insert (xi, ta)
ta=0
ta= ∞
Reflect
λ=reflection
ta=0
ta=time delay
dext CTRL/ update queue ta
ta=0
Update Polarization
λ=ctrl msg
dext ENV/ update queue ta; set ta
dint/ needRespond =Y
ta=0
ta= time delay
dint/ interrrupt =Y; needRespond=Y; set ta
dint/ get queue(min); set ta
ta=time delay
Respond
λ=propagation
ta=0
dext OPT/ update queue ta; insert (xi,ta)
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 156. PM phase transition diagram.

P.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Optical packet properly attenuated and rotated to the limit of accuracy

- Optical packet propagated out the correct port at the correct time

P.9.3 Respond to an Environmental Packet in the Polarization Modulator (PM)

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
  - None

P.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

P.10 Polarization Modulator Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change

in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 6. *Polarization Modulator Test Cases.*

		Inject Ports					Running Totals		
Phase	**Case**	**Opt1**	**Opt2**	**Ctrl**	**Env**	**Notes**	opt #	**env #**	**ctrl #**
Passive	1	1	0	0	0	single	1	0	0
	2	0	1	0	0	single	2	0	0
	3	0	0	1	0	single	2	0	1
	4	0	0	0	1	single	2	1	1
	5	1	1	0	0	same time	4	1	1
	6	1	0	1	0	same time	5	1	2
	7	1	1	0	0	differ time	7	1	2
	8	1	0	1	0	differ time	8	1	3
	9	1	1	1	1	same time	10	2	4
	10	1	1	1	1	differ time	12	3	5
	11	0	1	0	1	same time	13	4	5
	12	0	1	0	1	differ time	14	5	5
	13	0	0	1	1	same time	14	6	6
	14	0	0	1	1	differ time	14	7	7
	15	1	0	0	1	same time	15	8	7
	16	1	0	0	1	differ time	16	9	7
	20	2	0	0	0	same time	18	9	7
	21	0	2	0	0	same time	20	9	7
	22	2	1	0	0	same time	23	9	7

	23	2	0	1	0	same time	25	9	8
	24	2	0	0	1	same time	27	10	8
	25	2	0	1	0	differ time	29	10	9
	26	2	1	1	1	same time	32	11	10
	27	2	1	1	1	differ time	35	12	11
	28	0	2	0	1	same time	37	13	11
	29	0	2	0	1	differ time	39	14	11
	30	0	0	1	1	same time	39	15	12
	31	0	0	1	1	differ time	39	16	13
	32	2	0	0	1	same time	41	17	13
	33	2	0	0	1	differ time	43	18	13
totals		27	16	13	18				
Respond	41	2	0	0	0	single	45	18	13
	42	1	1	0	0	single	47	18	13
	43	1	0	1	0	single	48	18	14
	44	1	0	0	1	single	49	19	14
	45	2	1	0	0	same time	52	19	14
	46	2	0	1	0	same time	54	19	15
	47	2	0	0	1	differ time	56	20	15
	48	2	0	1	0	differ time	58	20	16
	49	2	1	1	1	same time	61	21	17
	50	2	1	1	1	differ time	64	22	18
	51	1	1	0	1	same time	66	23	18
	52	1	1	0	1	differ time	68	24	18
	60	3	0	0	0	same time	71	24	18
	61	1	2	0	0	same time	74	24	18
	62	3	1	0	0	same time	78	24	18
	63	3	0	1	0	same time	81	24	19
	64	3	0	0	1	same time	84	25	19
	65	3	0	1	0	differ time	87	25	20
	66	3	1	1	1	same time	91	26	21
	67	3	1	1	1	differ time	95	27	22
	68	1	2	0	1	same time	98	28	22
	69	1	2	0	1	differ time	101	29	22
totals		43	15	9	11				
	TC1	1	0	1	2	single	102	31	23
	TC2	1	0	1	2	single	103	33	24
	TC3	1	0	1	2	single	104	35	25
	TC4	1	0	1	2	single	105	37	26
	TC5	1	0	1	2	single	106	39	27
	TC6	1	0	1	2	single	107	41	28
	TC7	1	0	1	2	single	108	43	29

	TC8	1	0	1	2	single	109	45	30
totals		8	0	8	16				

Notes: 23 - INIT control message sent; OPT1 & Ctrl - same time - Passive: downstream received packets = 214

25 - Set H control message sent - OPT1 & Ctrl - differ time - Passive: downstream received packets = 207, sent value of 2.1

26 - Set V control message sent - OPT1, OPT2, Ctrl & ENV - same time - Passive: downstream received packets = 207, sent value of 4, exceeds PI

27 - Set A control message sent - OPT1 & Ctrl - differ time - Passive: downstream received packets = 207, sent value of 1

30 - INIT control message sent - Ctrl & ENV - same time - Passive: downstream received packets = 214

46 - Set D control message sent - OPT1 & Ctrl - same time - Passive: downstream received packets = 207, sent value of -2

48 - Set angle control message sent - OPT1 & Ctrl - differ time - Passive: downstream received packets = 207, sent value of 4.5

50 - Get angle control message sent - OPT1 & Ctrl - differ time - Passive: downstream received packets = 207, should return PI

63 - INIT control message sent - OPT1 & Ctrl - same time - Respond: downstream received packets = 211

67 - INIT control message sent - OPT1, OPT2, Ctrl & ENV - differ time - Respond: downstream received packets = 207

P.11 References

None

Appendix Q - Polarizing Beamsplitter

Q.1 Device Description:

The polarizing beamsplitter (PBS) is an optical device used to split a beam of light into two orthogonally polarized outputs, or to combine two input streams into one output stream. In a "free-space" beamsplitter, light from one input fiber is sent through a collimating lens, then split into two orthogonal states and focused on the output port lenses. Fiber-only beamsplitters exist, but are wavelength-dependent and used in situations where the optical fiber carries a single wavelength. Polarization maintaining fibers are used on the main output ports and are aligned to maintain the polarization supplied by the splitter. See Figure 1 for orientation diagram.

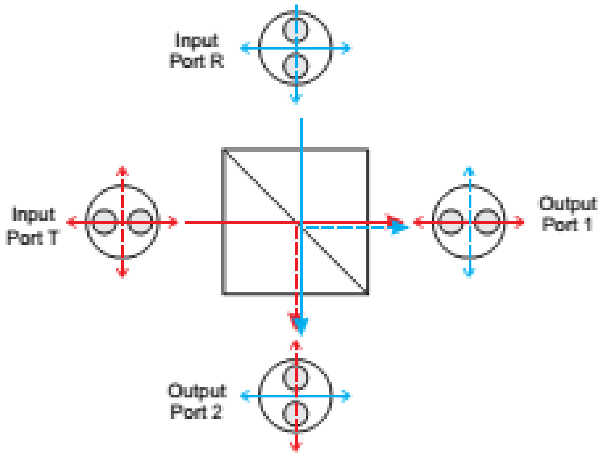


Figure 157. Standard orientation of polarization maintaining fibers on a PBS (OZOptics, 2013).

A PBS can be made from housing with collimating lenses and some form of a beam splitting material, usually a partially reflective mirror, which splits the light (Saleh & Teich, 1991) or can be fashioned from two triangular birefringent materials glued together. Physical designs include cubes mounted into brackets for free-space optics and housings that have permanently mounted pigtails or connectors for fiber lines. The amount of light directed to each port depends on the polarization of the incoming light. For example, in Figure 1 horizontal light input to Port T will pass through the device, with a very small amount passed to Port 2. Vertical light entering Port T will pass to Port 2 with a very small amount passing to Port 1. In the specific case of diagonal or antidiagonal light 50% of the power with be passed to each port. See Figure 1 for an example of a four port fiber-based PBS.

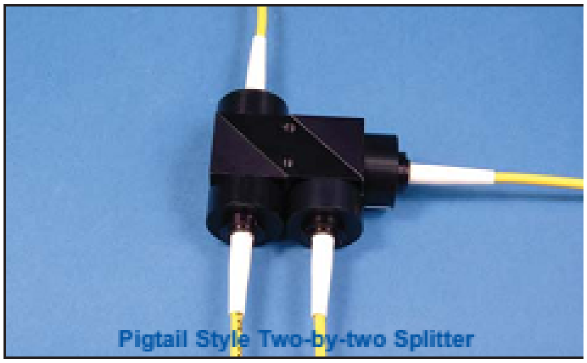


Figure 158. View of a two by two polarizing beamsplitter (OZOptics, 2013).

Although PBS may have from three to eight or more ports, this research will use the four port PBS devices with fiber pigtails, per the discussion with the SME.

The PBS is a bidirectional optical component with four optical ports. Light entering the primary port is split to exit two output ports with the splitting ratio dependent on the polarization of the incoming light. In the opposite direction, the component works the same way, splitting the light by passing through a portion of the beam and reflecting the rest to the port opposite the reflected port used by the primary path. The light suffers both a slight attenuation from the material of the device and the splitting medium has a phase effect for the reflected portion of the beam. The reflected beam undergoes a global phase shift of $\pi/2$ and is applied to light passing through the device in both directions.

The internal material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the PBS may become permanently damaged which changes its propagation characteristics. Similarly, the PBS is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the PBS may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the PBS is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the PBS. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the PBS to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the PBS using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the PBS. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

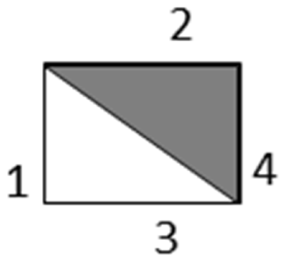


Figure 159. Symbol for the 4-port PBS in the QKD system architecture.

Q.2 Polarizing Beamsplitter Conceptual Model

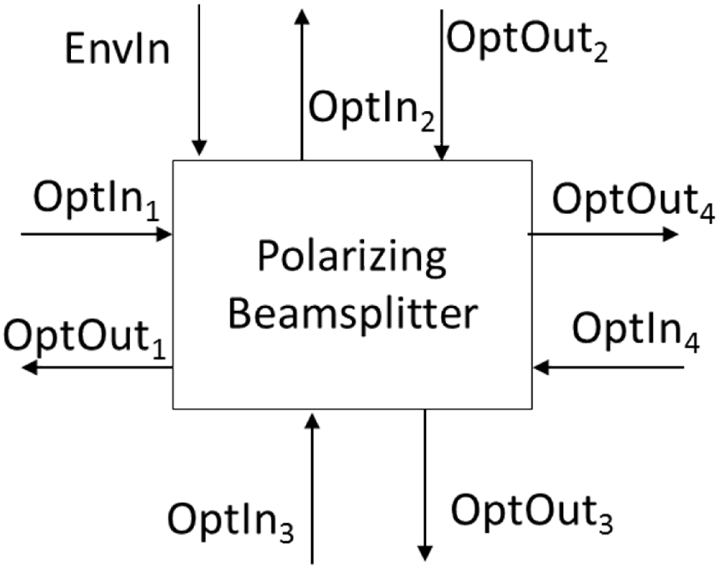


Figure 160. PBS conceptual model.

The conceptual model for a PBS consists of four optical input ports {OptIn$_1$, OptIn$_2$, OptIn$_3$, OptIn$_{4}$$_}$, four optical output ports {OptOut$_1$, OptOut$_2$, OptOut$_3$, OptOut$_4$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the PBS. In comparison to the PBS symbol used in the QKD simulation architecture shown in Figure 3, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the PBS, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 4 will instantaneously generate a reflected emitting out of OptOut_1.

The PBS calculates changes to the power (attenuation) of any packet coming through an optical port after a time equaling the propagation delay of the module. The packet is calculated at full power minus some small amount to account for attenuation through the device. The model splits each incoming optical packet into a 'passed' packet and a 'reflected' (the DEVS code in Section 1.7 uses 'strong' and 'weak'), with the strength of each packet determined by the packet polarization, and injects these packets into the queue. Each of these entries are a (port, value) pair, just as any other entry into the queue, with the [port] entry equal to the output port and the [value] equal to the adjusted values of the incoming packet. Additionally, packets output on the reflected port have $\pi/2$ added to their global phase (i.e. $\theta = \theta + \pi/2$) due to reflection off of the beamsplitting material inside the device. The outputs will be in a state determined by their polarization, in the case of Figure 1, packets input on Port T and exiting Port 1 will be in the state (α=0, ϕ=0), packets exiting Port 2 will be in the state (α= π /2, $\phi = 0$).

The PBS must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the PBS can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

Q.3 Mathematical Model

For a detailed mathematical description of the PBS, refer to Section 15.8 which contains the Mathematica worksheet provided by the optical physics SME.

Q.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the PBS.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Determine the input port number.
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Remove the packet from the queue and split it into two packets
- Update the values for one packet as a ‘strong’ optical signal based on the characteristics of the PBS, the original values of the input optical signal and the current environment and set the correct output port.

- Update the values for the other packet as a ‘weak’ optical signal based on the characteristics of the PBS, the original values of the input optical signal and the current environment and set the correct output port.
- Send the attenuated output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

Q.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the PBS in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the PBS at the same time.

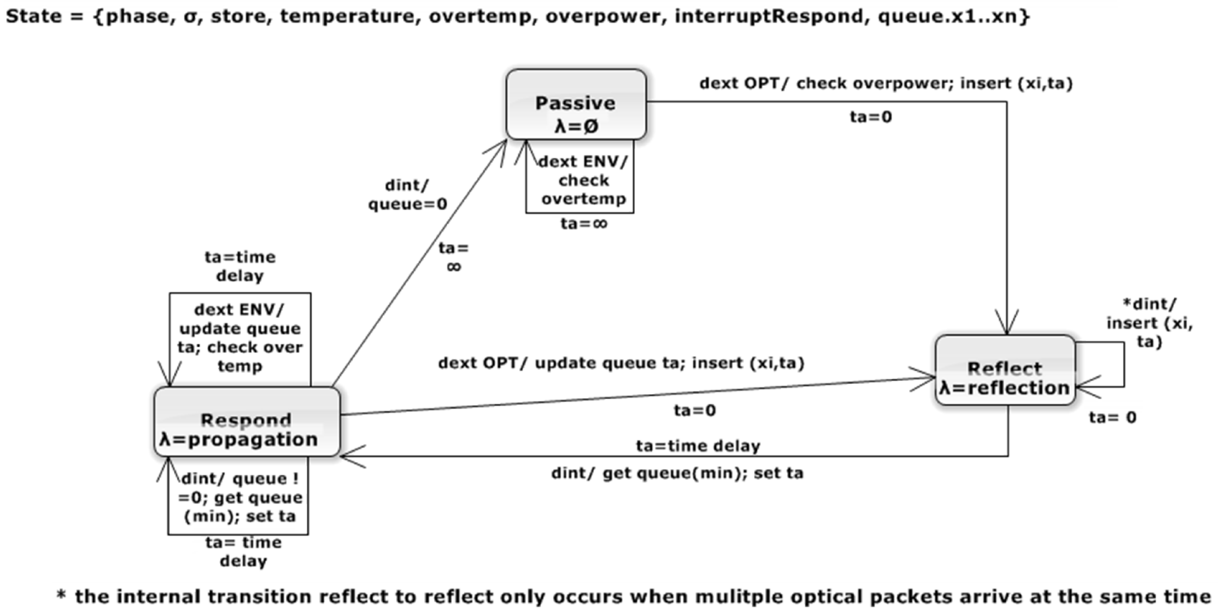

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/
check
overtemp
ta=∞
dint/
queue=0
ta=
∞
ta=time
delay
dext ENV/
update queue
ta; check over
temp
*dint/
insert (xi,
ta)
Reflect
λ=reflection
ta= 0
dext OPT/ update queue ta; insert (xi,ta)
ta=0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue !
=0; get queue
(min); set ta
ta= time
delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 161. PBS phase transition diagram.

Q.6 Event-Trace Diagram

This section shows various examples of packets entering the PBS. The tables list the states the PBS proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the PBS, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state

- Notes: any notes for that state

Q.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 66. *Case I state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	
5	s3	entry	passive	inf	x1	c	n	n	n	null	

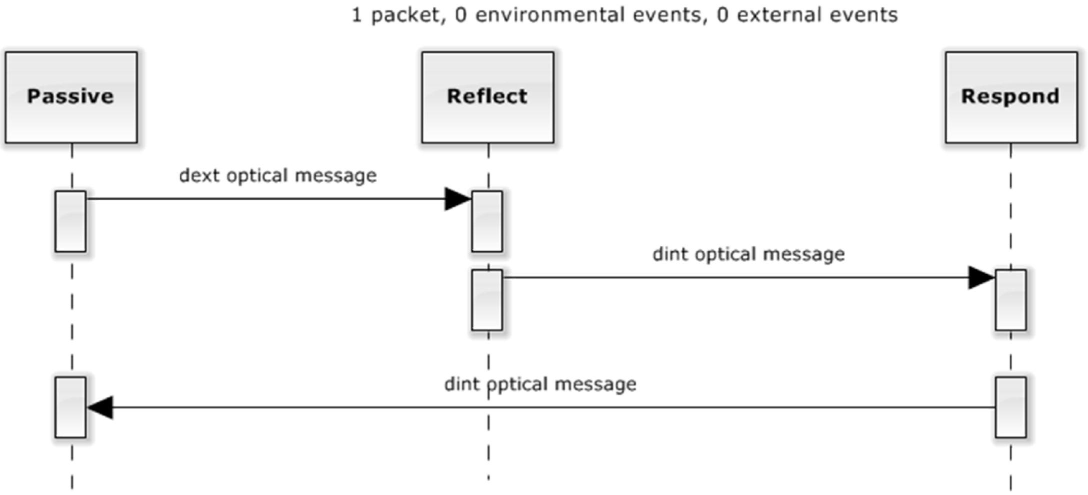


Figure 162. Case I sequence diagram.

Q.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 67. *Case II state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	Interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	

0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

1 packet, 0 environmental events, 1 external event (with 1 packet) at e=2

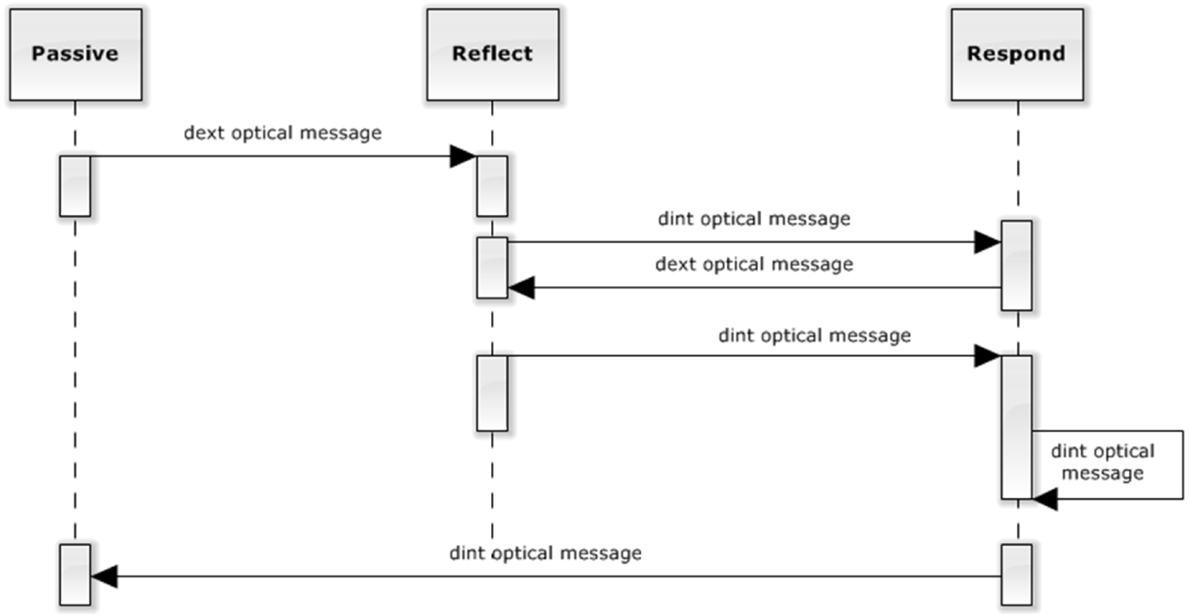


Figure 163. Case II sequence diagram.

Q.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 68. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	

0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

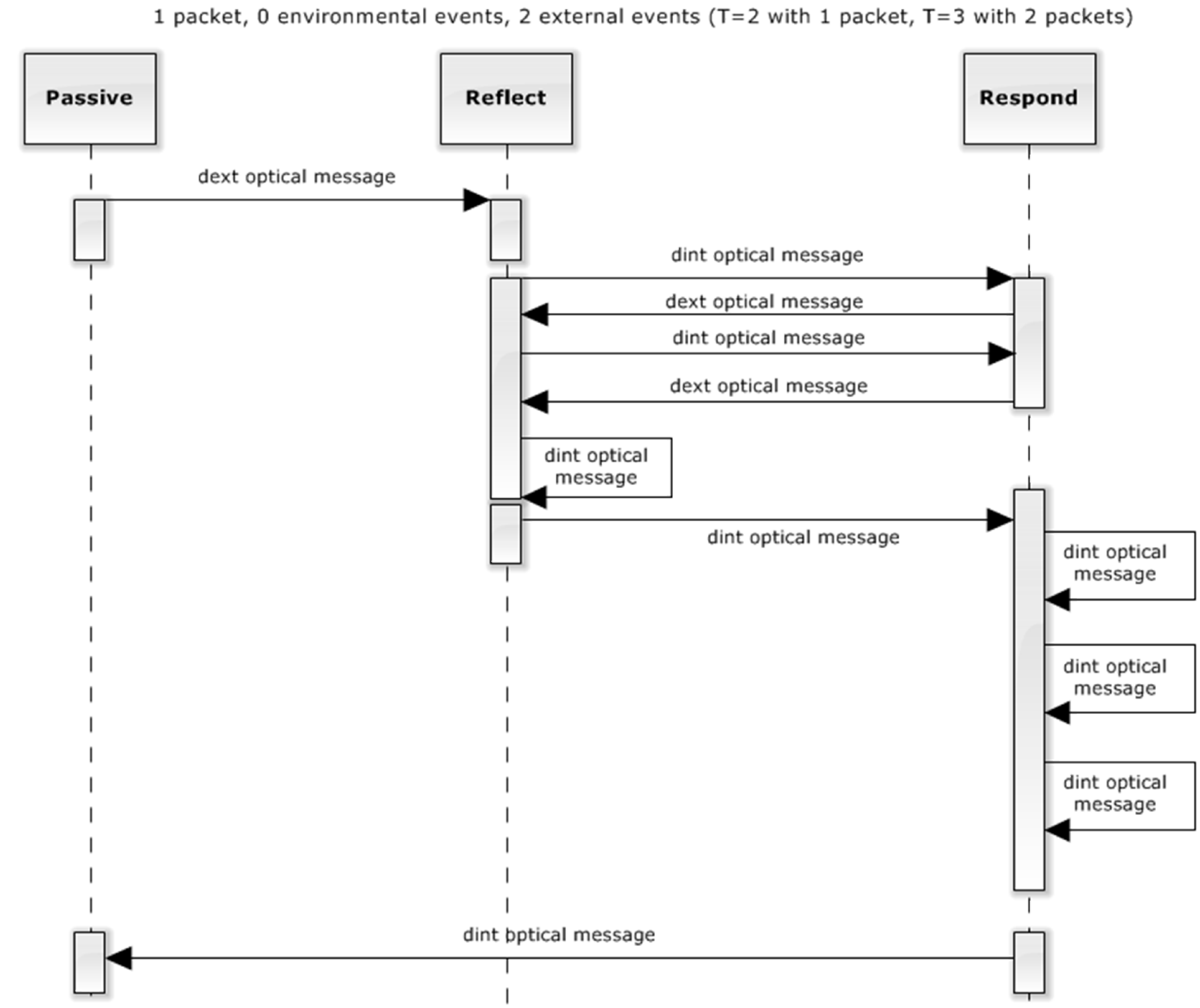


Figure 164. Case III sequence diagram.

Q.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 69. *Case IV state list.*

											Notes:
		entry/			store		over	over	interrupt	queue	assume
time	state	exit	phase	sigma	(*xi*)	temp	temp	power	respond	(*xi*, *tp*)	tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

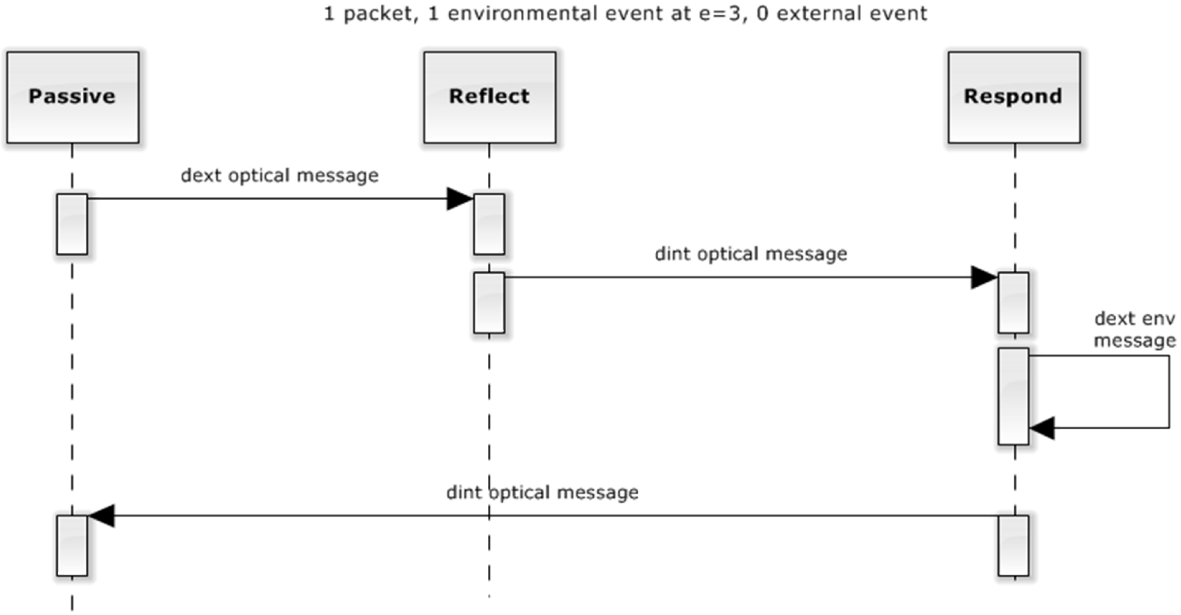


Figure 165. Case IV sequence diagram.

Q.7 Polarizing Beamsplitter Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the PBS we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \text{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS$_{PBS}$* = (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “reflect”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet

messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time $delay_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPassed() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcReflected() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*current.v*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
$q.v_{min}$ = minimum value in the queue

v.q = value from a queue entry

Every δ_{ext} puts all of its x (p,v) values into the variable *store*

InPorts = {“OptIn_1”, “OptIn_2”, “OptIn_3”, “OptIn_4”, “EnvIn”} with
X_M = {(“OptIn_1”, V_{opt}), (“OptIn_2”, V_{opt}), (“OptIn_3”, V_{opt}), (“OptIn_4”, V_{opt}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“OptOut_1”, “OptOut_2”, “OptOut_3”, “OptOut_4”} with
Y_M = {(“OptOut_1”, $Y_{OptOut1}$), (“OptOut_2”, $Y_{OptOut2}$), (“OptOut_3”, $Y_{OptOut3}$), (“OptOut_4”, $Y_{OptOut4}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond, queue*} = {{“passive”, “reflect”, “respond”} x R_0^+ x V x R x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x V}

External Transition Function:

```
δ_ext(phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue, e, ((p_i,v_i),….
                                                                                      (p_n,v_n))) =
(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = “passive” and p ∈ {“OptIn_1”, “OptIn_2”, “OptIn_3”}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower =  “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
if phase = “respond” and p ∈ {“OptIn_1”, “OptIn_2”, “OptIn3”}
   update_delay(queue)
    for messagebag != null
     current = messagebag_first()
     if current.value.power > damaged.power
       overpower =  “Y”
     insert_event_q(current)
      remove_event_m(current)
   queue.current = queue_need_reflected()
```

```
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond= “Y”
   timeLeftRespond = timeLeftRespond - e

(“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                   queue.x1..xn)
   if phase = “respond” and p = “EnvIn”
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”
     time.delay = timeLeftRespond

(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
(“reflect”, 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
   if phase = “reflect” and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)

 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
   if phase = “reflect” and need.reflect = null
     need.reflect = queue_need_reflected()
     if interruptRespond = “N”
       current = queue_min()
       time.delay = current.time.delay
     if current.p = “$\text{OptIn}_1$”              /* input port 1 – strong 4 weak 3 */
        new1 = (“$\text{OptOut}_3$”,calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
        new2 = (“$\text{OptOut}_4$”,calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
     else
        if current.p = “$\text{OptIn}_2$”            /* input port 2 – strong 3 weak 4 */
```

```
      new1 = ("OptOut$_4$",calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
      new2 = ("OptOut$_3$",calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
  else
    if current.p  = "OptIn$_3$"            /* input port 3 – strong 2 weak 1 */
      new1 = ("OptOut$_1$",calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
      new2 = ("OptOut$_2$",calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
    else                                   /* input port 4 – strong 1 weak 2*/
      new1 = ("OptOut$_2$",calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
      new2 = ("OptOut$_1$",calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
 timeLeftRespond = propagation delay
 else
  time_delay = timeLeftRespond

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                    queue.x1..xn)
 if phase = "respond" and size > 0
  update_delay(queue)
  size= queue_size()
  current = queue_min()
  time.delay = current.time.delay
  if current.p  = "OptIn$_1$"              /* input port 1 – strong 4 weak 3 */
    new1 = ("OptOut$_3$",calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
    new2 = ("OptOut$_4$",calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
  else
    if current.p  = "OptIn$_2$"            /* input port 2 – strong 3 weak 4 */
      new1 = ("OptOut$_4$",calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
      new2 = ("OptOut$_3$",calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
  else
    if current.p  = "OptIn$_3$"            /* input port 3 – strong 2 weak 1 */
      new1 = ("OptOut$_1$",calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
      new2 = ("OptOut$_2$",calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
    else                                   /* input port 4 – strong 1 strong 2*/
      new1 = ("OptOut$_2$",calcPassed(current.v, temperature, overtemp, peakpwr, overpwr))
      new2 = ("OptOut$_1$",calcReflected(current.v, temperature, overtemp, peakpwr, overpwr))
   interruptRespond= "N"

("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
 if phase = "respond" and size  = 0
  size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

λ(*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, queue*) =

(*reflect.p, reflect.v*)
  if phase = “reflect”

(*new1.p*, *new1.v*)
  if phase = “respond”

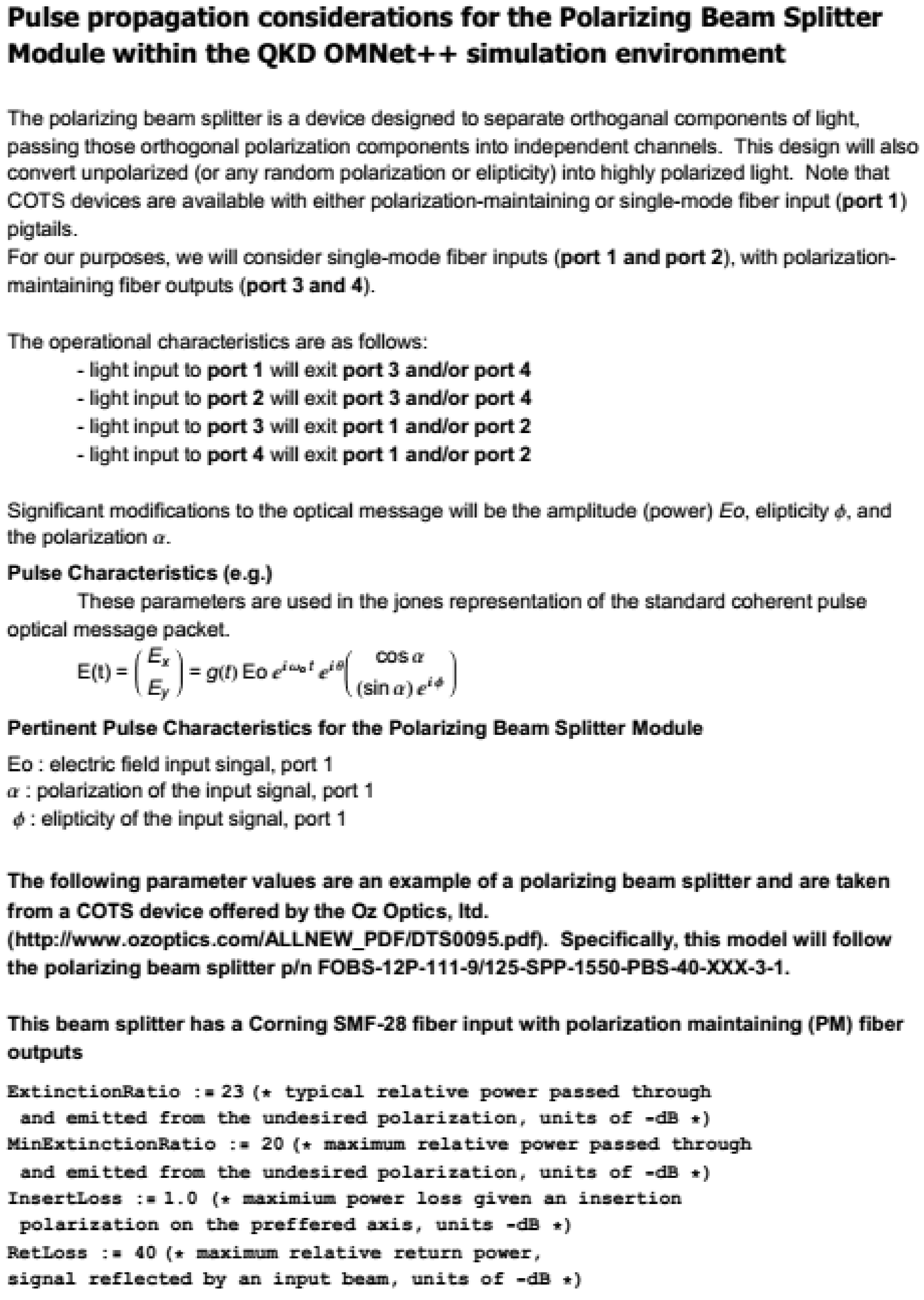
(*new2.p*, *new2.v*)
  if phase = “respond”

∅ (null output)
  otherwise;

Time advance Function:

ta(*phase*, *σ*, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, queue*) = *σ*;

Q.8 Mathematical Model



Pulse propagation considerations for the Polarizing Beam Splitter Module within the QKD OMNet++ simulation environment

The polarizing beam splitter is a device designed to separate orthoganal components of light, passing those orthogonal polarization components into independent channels. This design will also convert unpolarized (or any random polarization or elipticity) into highly polarized light. Note that COTS devices are available with either polarization-maintaining or single-mode fiber input (**port 1**) pigtails.
For our purposes, we will consider single-mode fiber inputs (**port 1 and port 2**), with polarization-maintaining fiber outputs (**port 3 and 4**).

The operational characteristics are as follows:

- light input to **port 1** will exit **port 3 and/or port 4**
- light input to **port 2** will exit **port 3 and/or port 4**
- light input to **port 3** will exit **port 1 and/or port 2**
- light input to **port 4** will exit **port 1 and/or port 2**

Significant modifications to the optical message will be the amplitude (power) *Eo*, elipticity ϕ, and the polarization α.

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\, e^{i\omega_0 t}\, e^{i\theta} \begin{pmatrix} \cos\alpha \\ (\sin\alpha)\, e^{i\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Polarizing Beam Splitter Module

Eo : electric field input singal, port 1
α : polarization of the input signal, port 1
ϕ : elipticity of the input signal, port 1

The following parameter values are an example of a polarizing beam splitter and are taken from a COTS device offered by the Oz Optics, ltd. (http://www.ozoptics.com/ALLNEW_PDF/DTS0095.pdf). Specifically, this model will follow the polarizing beam splitter p/n FOBS-12P-111-9/125-SPP-1550-PBS-40-XXX-3-1.

This beam splitter has a Corning SMF-28 fiber input with polarization maintaining (PM) fiber outputs

```
ExtinctionRatio := 23 (* typical relative power passed through
 and emitted from the undesired polarization, units of -dB *)
MinExtinctionRatio := 20 (* maximum relative power passed through
 and emitted from the undesired polarization, units of -dB *)
InsertLoss := 1.0 (* maximium power loss given an insertion
 polarization on the preffered axis, units -dB *)
RetLoss := 40 (* maximum relative return power,
signal reflected by an input beam, units of -dB *)
```

Example Initial Conditions.
Evaluate this cell to see the results for a given input polarization; do not evaluate this cell if you wish to see the general forms of the following functions :

$$\alpha := \pi / 4$$
$$\phi := 0$$
$$\theta := 0$$

$$\texttt{OptVect}[\alpha\_,\ \phi\_,\ \theta\_] := g\,Eo\ e^{i\,\omega_o\,t}\ e^{i\,\theta} \begin{pmatrix} \texttt{Cos}[\alpha] \\ \texttt{Sin}[\alpha]\ e^{i\,\phi} \end{pmatrix}$$

We define the horizontal lab frame as γ=0 (vertical as γ=π/2). The following chart describes the behavior of the four-port beam splitter.

Input Port	Passed Polarization	Reflected Polarization
1	4 ($\gamma = 0$)	3 ($\gamma = \pi / 2$)
2	3 ($\gamma = 0$)	4 ($\gamma = \pi / 2$)
3	2 ($\gamma = 0$)	1 ($\gamma = \pi / 2$)
4	1 ($\gamma = 0$)	2 ($\gamma = \pi / 2$)

Note here that "reflection" refers to the behavior of the opitcal beam impinging upon the beam splitter; it does not refer to reflection in the opposite in direction from and on the same fiber as that of the input optical beam. For the orthogonal output polarizations, γ will be used to determine the output polarization, 0 and π/2 for the output ports. The second, additive term is to include the non - ideality of the polarizing beam splitter - it allows a small portion of the power, set by "ExtinctionRatio", to be passed through in the "blocked" polarization.

$$\texttt{Polarizer}[\gamma\_] := \begin{pmatrix} (\texttt{Cos}[\gamma])^2 & (\texttt{Cos}[\gamma] * \texttt{Sin}[\gamma]) \\ (\texttt{Cos}[\gamma] * \texttt{Sin}[\gamma]) & (\texttt{Sin}[\gamma])^2 \end{pmatrix} + \sqrt{10^{-\texttt{ExtinctionRatio}/10}} \begin{pmatrix} (\texttt{Sin}[\gamma])^2 & (-\texttt{Cos}[\gamma] * \texttt{Sin}[\gamma]) \\ (-\texttt{Cos}[\gamma] * \texttt{Sin}[\gamma]) & (\texttt{Cos}[\gamma])^2 \end{pmatrix}$$

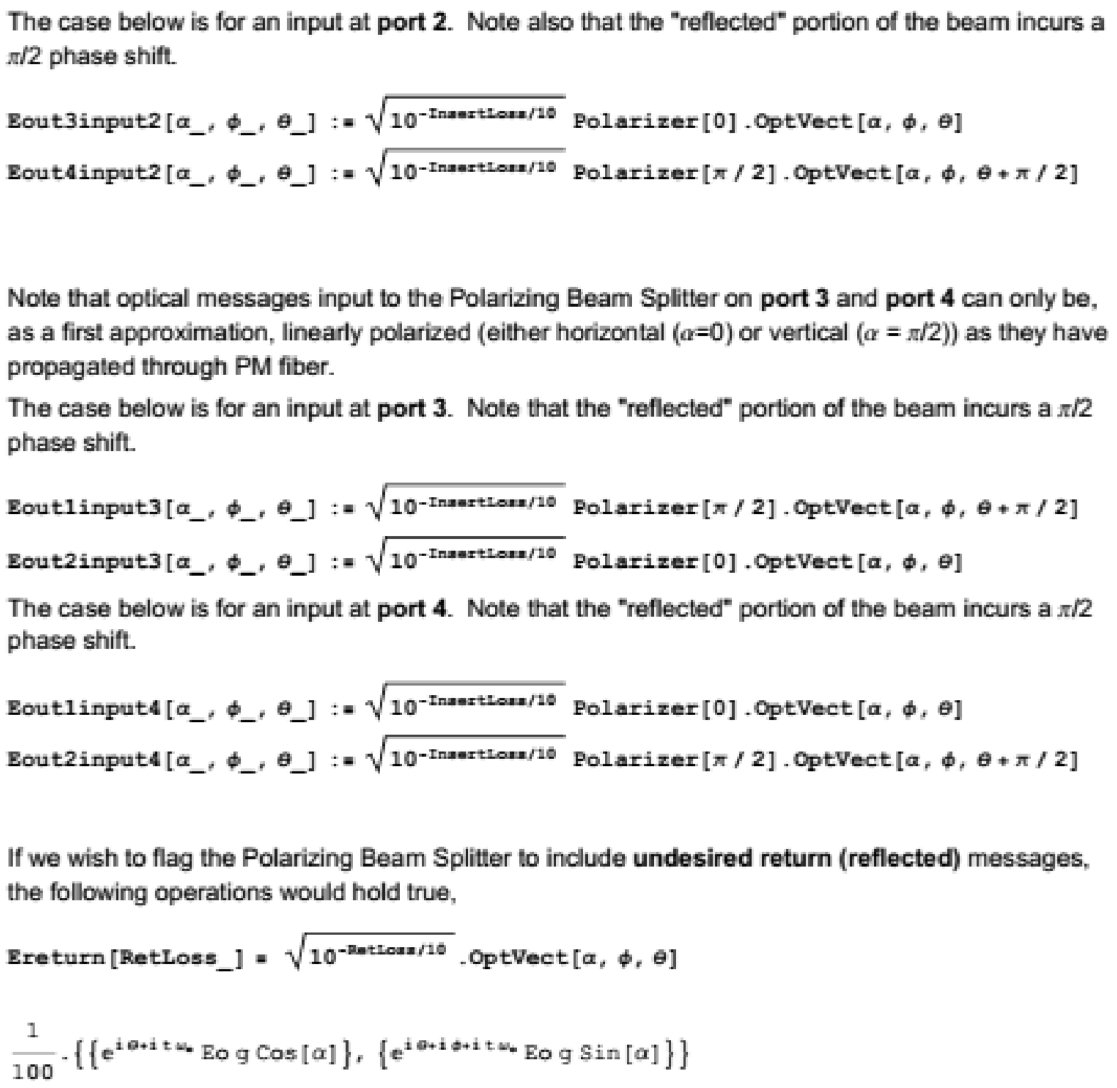
Note that outputs at **port 3** and **port 4** will be propagating through PM fiber. As such, the two orthogonal polarizations will propagate at distinct velocities, and should be treated as independent opical messages.

The case below is for an input at **port 1.** Note also that the "reflected" portion of the beam incurs a π/2 phase shift.
Following cases will have outputs hidden (remove the ":" before the "=" to un-hide outputs).

$$\texttt{Eout3input1}[\alpha\_,\ \phi\_,\ \theta\_] = \sqrt{10^{-\texttt{InsertLoss}/10}}\ \texttt{Polarizer}[\pi / 2].\texttt{OptVect}[\alpha,\ \phi,\ \theta + \pi / 2]\ \texttt{// MatrixForm}$$

$$\texttt{Eout4input1}[\alpha\_,\ \phi\_,\ \theta\_] = \sqrt{10^{-\texttt{InsertLoss}/10}}\ \texttt{Polarizer}[0].\texttt{OptVect}[\alpha,\ \phi,\ \theta]\ \texttt{// MatrixForm}$$

$$\begin{pmatrix} 0.0630957\ e^{i\left(\frac{\pi}{2}+\theta\right)+i\,t\,\omega_o}\ Eo\ g\ \texttt{Cos}[\alpha] \\ 0.891251\ e^{i\left(\frac{\pi}{2}+\theta\right)+i\,\phi+i\,t\,\omega_o}\ Eo\ g\ \texttt{Sin}[\alpha] \end{pmatrix}$$



The case below is for an input at **port 2**. Note also that the "reflected" portion of the beam incurs a $\pi/2$ phase shift.

`Eout3input2[α_, ϕ_, θ_] :=` $\sqrt{10^{-\text{InsertLoss}/10}}$ `Polarizer[0].OptVect[α, ϕ, θ]`

`Eout4input2[α_, ϕ_, θ_] :=` $\sqrt{10^{-\text{InsertLoss}/10}}$ `Polarizer[π / 2].OptVect[α, ϕ, θ + π / 2]`

Note that optical messages input to the Polarizing Beam Splitter on **port 3** and **port 4** can only be, as a first approximation, linearly polarized (either horizontal (α=0) or vertical (α = $\pi/2$)) as they have propagated through PM fiber.

The case below is for an input at **port 3**. Note that the "reflected" portion of the beam incurs a $\pi/2$ phase shift.

`Eout1input3[α_, ϕ_, θ_] :=` $\sqrt{10^{-\text{InsertLoss}/10}}$ `Polarizer[π / 2].OptVect[α, ϕ, θ + π / 2]`

`Eout2input3[α_, ϕ_, θ_] :=` $\sqrt{10^{-\text{InsertLoss}/10}}$ `Polarizer[0].OptVect[α, ϕ, θ]`

The case below is for an input at **port 4**. Note that the "reflected" portion of the beam incurs a $\pi/2$ phase shift.

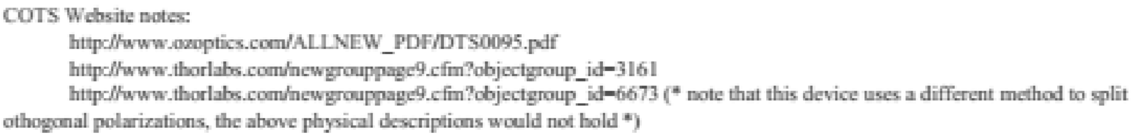
`Eout1input4[α_, ϕ_, θ_] :=` $\sqrt{10^{-\text{InsertLoss}/10}}$ `Polarizer[0].OptVect[α, ϕ, θ]`

`Eout2input4[α_, ϕ_, θ_] :=` $\sqrt{10^{-\text{InsertLoss}/10}}$ `Polarizer[π / 2].OptVect[α, ϕ, θ + π / 2]`

If we wish to flag the Polarizing Beam Splitter to include **undesired return (reflected)** messages, the following operations would hold true,

`Ereturn[RetLoss_] =` $\sqrt{10^{-\text{RetLoss}/10}}$ `.OptVect[α, ϕ, θ]`

$$\frac{1}{100}\cdot\left\{\left\{e^{i\theta+i t\omega_o}\,\text{Eo g Cos}[\alpha]\right\},\left\{e^{i\theta+i\phi+i t\omega_o}\,\text{Eo g Sin}[\alpha]\right\}\right\}$$



COTS Website notes:

http://www.ozoptics.com/ALLNEW_PDF/DTS0095.pdf
http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=3161
http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6673 (* note that this device uses a different method to split othogonal polarizations, the above physical descriptions would not hold *)

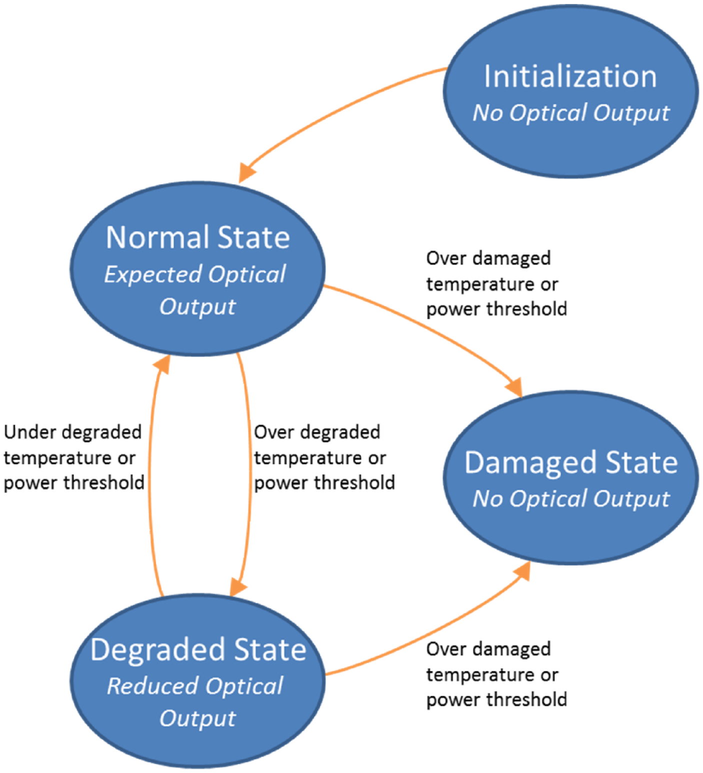
Q.9 Component Use Case

Q.9.1 Respond to an Optical Packet in the Polarizing Beam Splitter (PBS)

Optical packet arrives at the PBS. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the PBS. Records overpower condition, if applicable. Remove the optical packet from the queue and create a reflected and transmitted packet. Calculate the attenuated optical output signal based on the input signal, the output port and the current component state. Propagate the attenuated optical output signals out of the component optical ports based on the input port and whether transmitted or reflected.

- Identified Alternative Uses Cases
  - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 166. Component states.

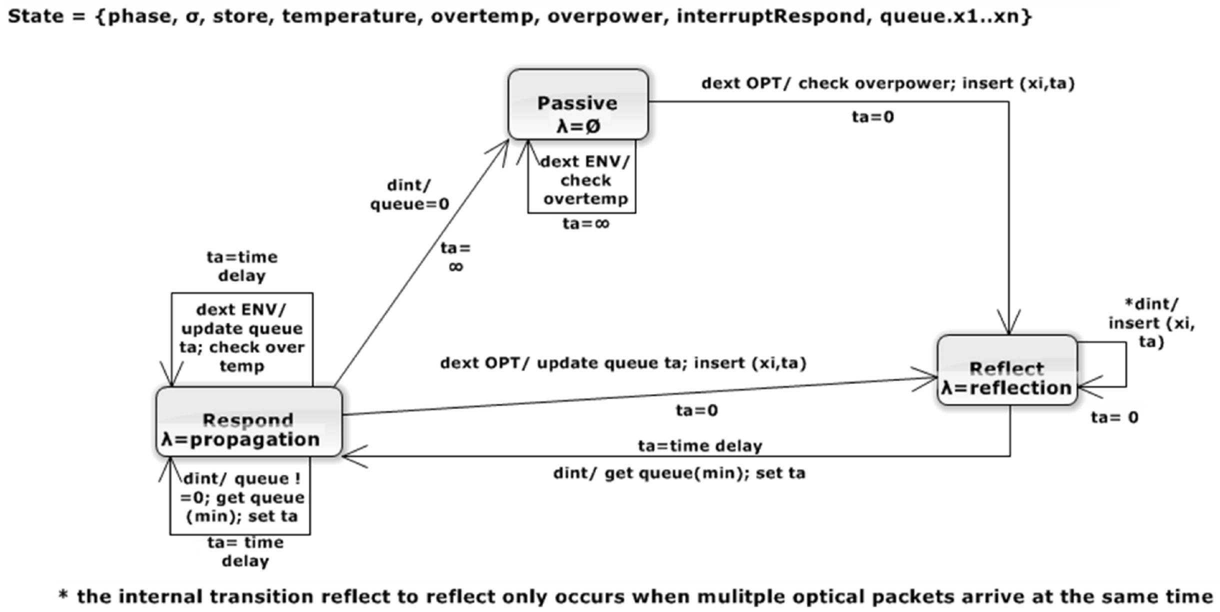

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
Respond
λ=propagation
dext OPT/ update queue ta; insert (xi,ta)
ta=0
Reflect
λ=reflection
*dint/ insert (xi, ta)
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue !=0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 167. PBS phase transition diagram.

Q.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Optical packet attenuated properly to the limit of accuracy
- Optical packet propagated out the correct port at the correct time

Q.9.3 Respond to an Environmental Packet in the Polarizing Beam Splitter (PBS)

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

Q.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary

- Change component function properly, if necessary

Q.10 PBS Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *PBS Test Cases*

		Inject Ports						Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Env**	**Notes**	opt #	**env #**
Passive	1	1	0	0	0	0	single	1	0
	2	0	1	0	0	0	single	2	0
	3	0	0	1	0	0	single	3	0
	4	0	0	0	1	0	single	4	0
	5	0	0	0	0	1	single	4	1
	6	1	1	1	1	0	same time	8	1
	7	1	1	1	1	0	differ time	12	1
	8	1	1	1	1	1	same time	16	2
	9	1	1	1	1	1	differ time	20	3
	10	0	1	0	0	1	same time	21	4
	11	0	1	0	0	1	differ time	22	5
	12	1	0	0	0	1	same time	23	6
	13	1	0	0	0	1	differ time	24	7
	14	0	0	1	0	1	same time	25	8
	15	0	0	1	0	1	differ time	26	9
	16	0	0	0	1	1	same time	27	10
	17	0	0	0	1	1	differ time	28	11
	20	2	0	0	0	0	same time	30	11
	21	0	2	0	0	0	same time	32	11
	22	0	0	2	0	0	same time	34	11
	23	0	0	0	2	0	same time	36	11
	24	2	2	2	2	0	same time	44	11
	25	2	2	2	2	0	differ time	52	11
	26	2	2	2	2	1	same time	60	12
	27	2	2	2	2	1	differ time	68	13
	28	0	2	0	0	1	same time	70	14
	29	0	2	0	0	1	differ time	72	15
	30	2	0	0	0	1	same time	74	16
	31	2	0	0	0	1	differ time	76	17
	32	0	0	2	0	1	same time	78	18
	33	0	0	2	0	1	differ time	80	19
	34	0	0	0	2	1	same time	82	20
	35	0	0	0	2	1	differ time	84	21
totals		21	21	21	21	21	84		
Respond	41	2	0	0	0	0	single	86	21
	42	0	2	0	0	0	single	88	21
	43	0	0	2	0	0	single	90	21
	44	0	0	0	2	0	single	92	21
	45	1	0	0	0	1	single	93	22

	46	2	1	1	1	0	same time	98	22
	47	2	1	1	1	0	differ time	103	22
	48	2	1	1	1	1	same time	108	23
	49	2	1	1	1	1	differ time	113	24
	50	0	2	0	0	1	same time	115	25
	51	0	2	0	0	1	differ time	117	26
	52	2	0	0	0	1	same time	119	27
	53	2	0	0	0	1	differ time	121	28
	54	0	0	2	0	1	same time	123	29
	55	0	0	2	0	1	differ time	125	30
	56	0	0	0	2	1	same time	127	31
	57	0	0	0	2	1	differ time	129	32
	60	3	0	0	0	0	same time	132	32
	61	0	3	0	0	0	same time	135	32
	62	0	0	3	0	0	same time	138	32
	63	0	0	0	3	0	same time	141	32
	64	3	2	2	2	0	same time	150	32
	65	3	2	2	2	0	differ time	159	32
	66	3	2	2	2	1	same time	168	33
	67	3	2	2	2	1	differ time	177	34
	68	0	3	0	0	1	same time	180	35
	69	0	3	0	0	1	differ time	183	36
	70	3	0	0	0	1	same time	186	37
	71	3	0	0	0	1	differ time	189	38
	72	0	0	3	0	1	same time	192	39
	73	0	0	3	0	1	differ time	195	40
	74	0	0	0	3	1	same time	198	41
	75	0	0	0	3	1	differ time	201	42
totals		36	27	27	27	21	117		
Math	TC1	1	0	0	0	2	single	202	44
	TC2	1	0	0	0	2	single	203	46
	TC3	1	0	0	0	2	single	204	48
	TC4	1	0	0	0	2	single	205	50
	TC5	1	0	0	0	2	single	206	52
	TC6	1	0	0	0	2	single	207	54
	TC7	1	0	0	0	2	single	208	56
totals		7	0	0	0	14	7		

Q.11 References

OZOptics. (2013). Beam splitters/combiners. Retrieved September 25, 2013, from http://www.ozoptics.com/ALLNEW_PDF/DTS0095.pdf

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

Appendix R - Single Mode Optical Fiber (SM Fiber)

R.1 Device Description:

Single mode fiber is used throughout optical components. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass. It has a core which guides the light and an outer cladding that reflects the internal light back into the core, bouncing the light down the fiber. This cladding helps to reflect outside light to keep in from entering the core. This structure allows for low loss over long distances (Saleh & Teich, 1991). The single-mode of the fiber comes from using a small core diameter (~10µm @ 1550nm) and small numerical aperture with the fundamental mode having a bell-shaped spatial distribution similar (Saleh & Teich, 1991; ThorLabs, 2013). See Figure 1 for an example of a single fiber cable.

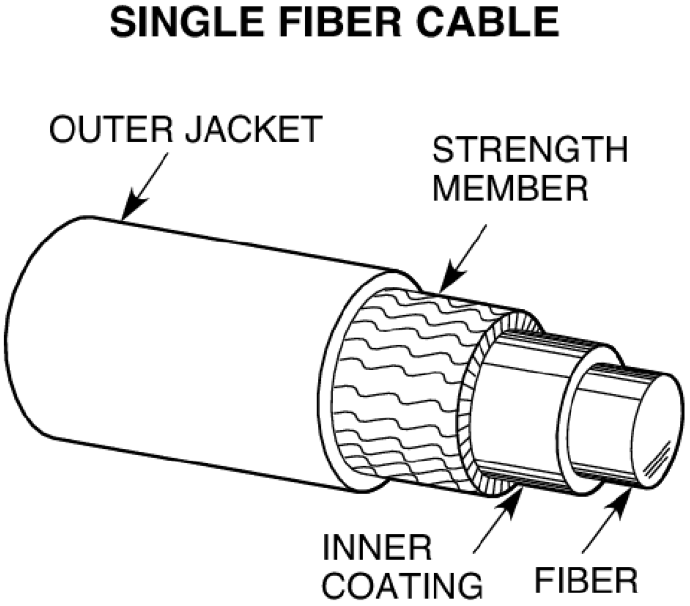


Figure 168. View of a single fiber cable (Newport, 2013).

The SM fiber is a bidirectional optical component with two optical ports. Light entering the primary port is propagated through the fiber, suffering both a slight attenuation from the material of the device and a small propagation delay dependent on the temperature of the fiber.

This type of fiber does not maintain polarization of the incoming light, so there will be a random polarization effect on the light.

The internal material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the SM fiber may become permanently damaged which changes its propagation characteristics. Similarly, the SM fiber is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the SM fiber may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the SM fiber is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. Optical propagation is complex process, starting with the characteristics of the optical fiber:

- Loss
- Index of refraction
- Zero-dispersion wavelengths
- Zero-dispersion slopes
- Coefficient of thermal expansion
- Chromatic dispersion
- Polarization mode dispersion
- Rayleigh backscatter

Other considerations include:

- Temperature
- Vibration and disturbance
- Out-band wavelengths
- Degraded or damaged fiber

- Raman scattering

These characteristics and considerations form a complex web of dependencies that the model must include if light is to be properly modeled. See Figure 2 for the dependency web from the optical SME.

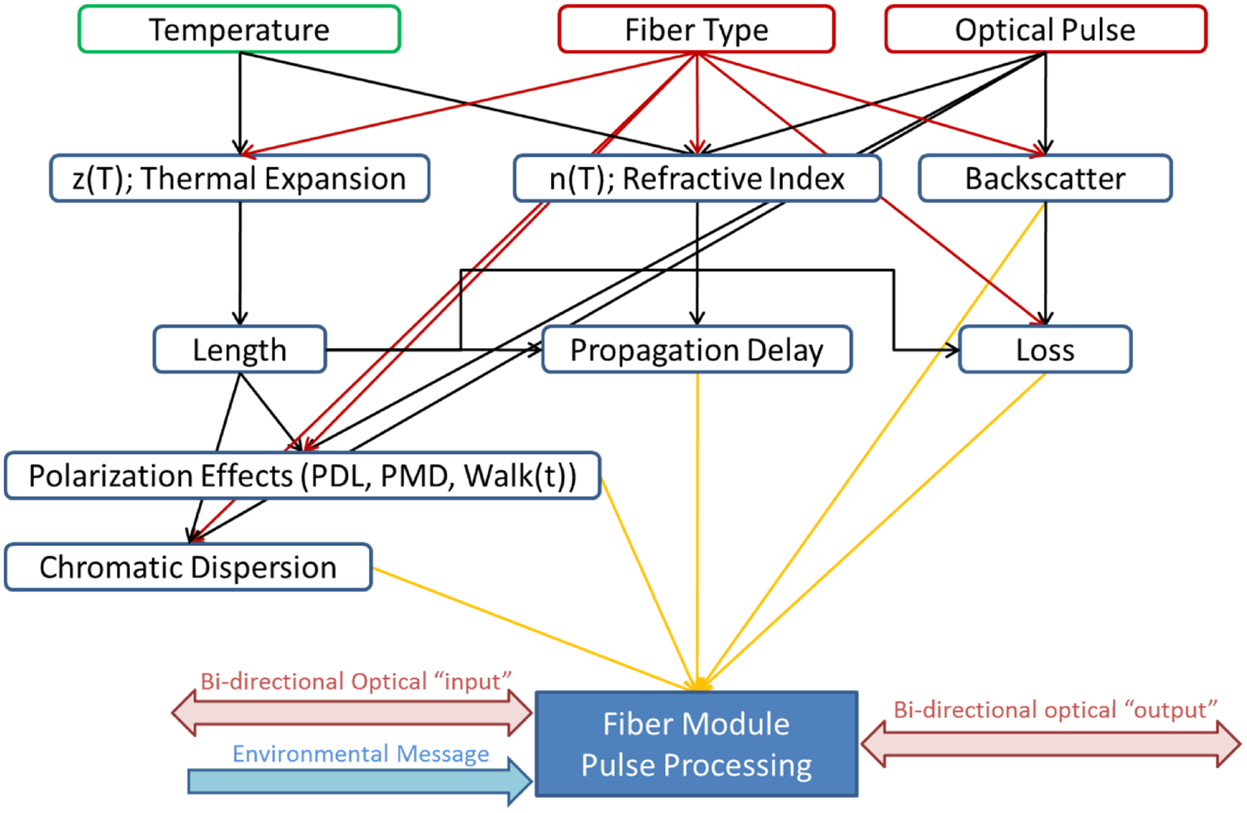


Figure 169. SMF-28 Physical component dependency web (Optical SME).

Modeling light as discrete event requires the modeler to make approximations for many of its characteristics, starting with the waveform. The optical model must completely describe any type of optical field (pulsed, continuous wave, arbitrary polarization states, etc.) and be adaptable for future changes. The optical SME created an optical light model that uses parameters derived from the Jones vector notation of light and uses a combination of three Gaussians envelopes. See Figure 1 for the Gaussian approximation of a laser source.

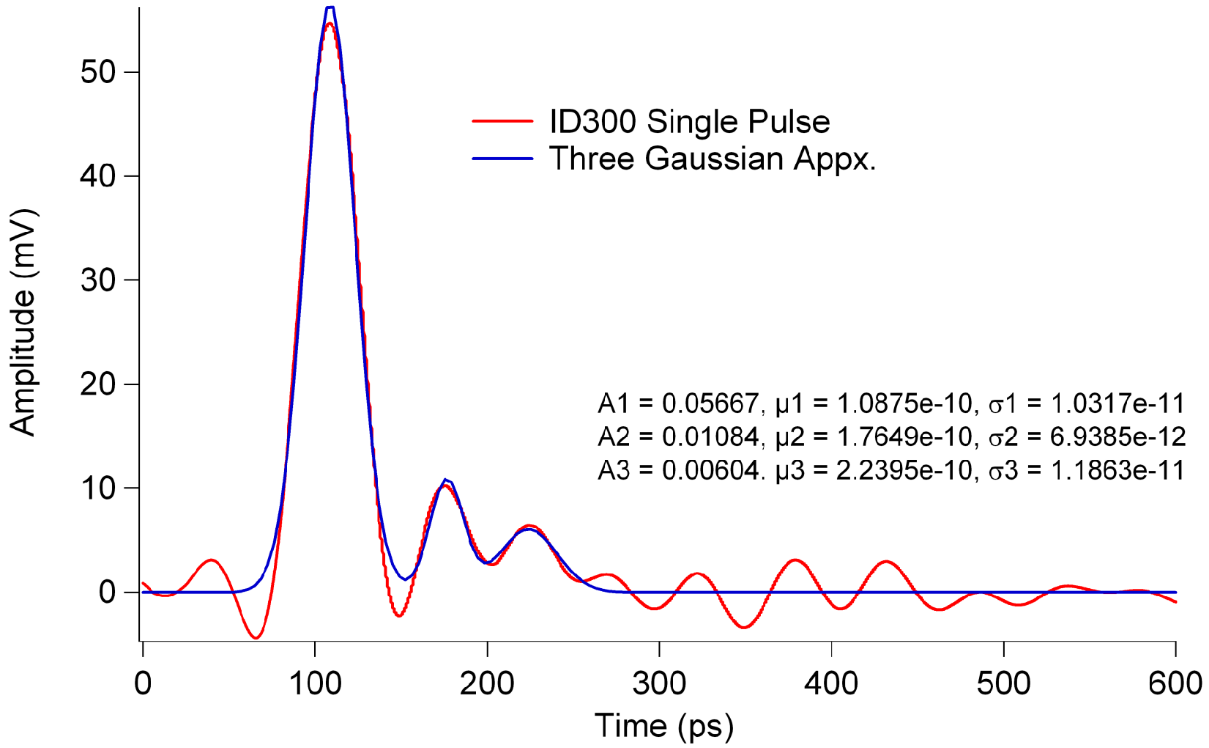


Figure 170. Approximation of an ID Quantique ID300 Pulsed laser source (Optical SME).

The environment surrounding a fiber is not static. As the environment and external stresses (temperature change, wind effects on aerial fibers, vibrations passed through the ground to buried cables etc.) change, the birefringent state of the fiber randomly changes over time. A random variation of the polarization mode coupling along the length of the fiber is induced as well. See Figure 4 for an example of the "polarization walk." See Appendix 2 for the mathematical model for this variation.

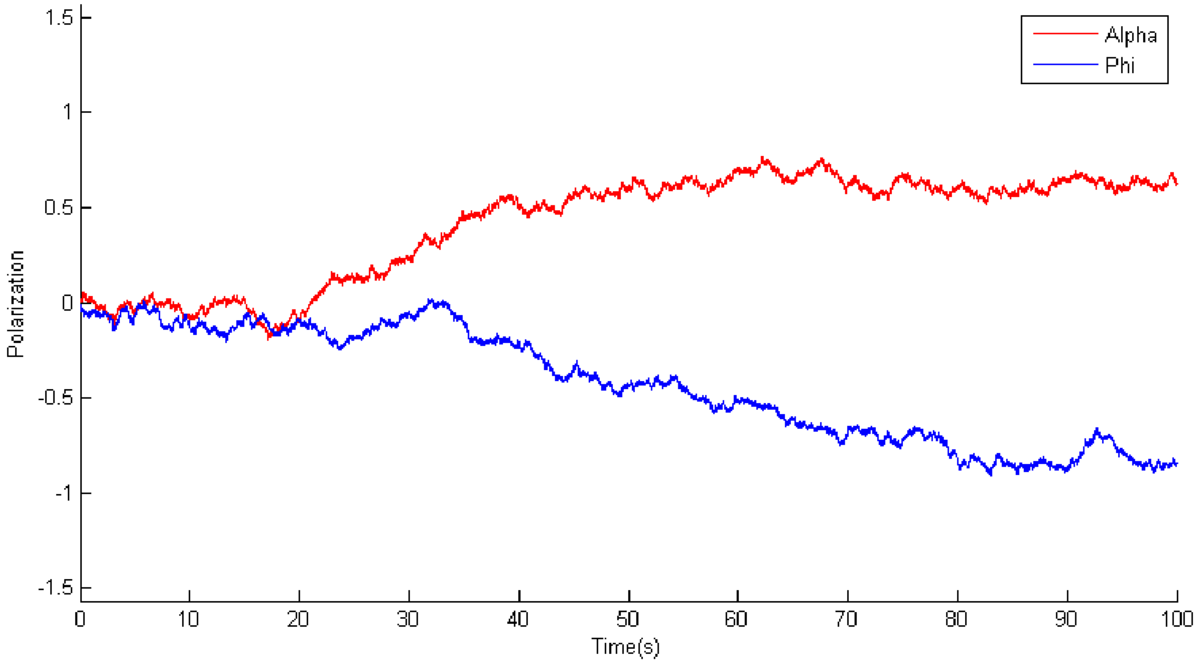


Figure 171. "Polarization walk" induced over time (optical SME).

The importance of modeling light correctly cannot be understated, as this “optical packet” is the base of the entire optical model. The optical SME described the requirements for the optical model as “all elements in the optical physical layer must be capable of properly handling any optical input state, react properly to environmental “inputs”, accept command and control messaging (if required), and must be as physically realistic as possible.”

The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the SM fiber, developed a series of use cases that exercised the functionality of the device over a wide variety of conditions, verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the SM fiber to the researcher. See Appendix 2 for the worksheet.

The next step of the modeling effort was to develop a conceptual model of the SM fiber using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the SM fiber. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

Figure 172. Symbol for Singe Mode Fiber in the QKD system architecture.

R.2 Single Mode Fiber Conceptual Model

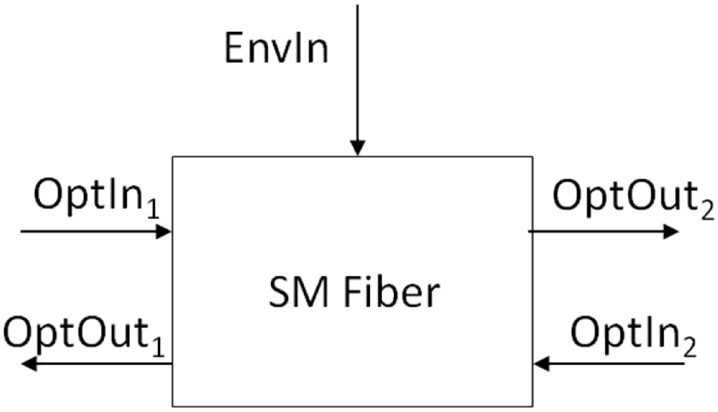


Figure 173. Single Mode fiber (SM fiber) conceptual model.

The conceptual model for a SM fiber consists of two optical input ports {OptIn_1, OptIn_2}, two optical output ports {OptOut_1, OptOut_2}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the SM fiber. In comparison to the SM fiber symbol used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the SM fiber, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The SM fiber must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the SM fiber can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the attenuation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation.

This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

R.3 Mathematical Model

For a detailed mathematical description of the SM fiber, refer to Section 16.8 which contains the Mathematica worksheet provided by the optical physics SME.

R.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the SM fiber.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.

- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.

- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Calculate the optical power of the signal. If the optical power is less than the minimum power, drop the pulse. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue.
- Remove the packet from the queue; calculate the attenuated output optical signal based upon the input optical signal, the OverPower flag, the OverTemp flag, and the current environment and calculate the delay through the fiber based on its length.
- Send the attenuated and delayed output signal out of the optical output port number that is not the same as the input port number.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.

- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

R.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the SM fiber in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the SM fiber at the same time.

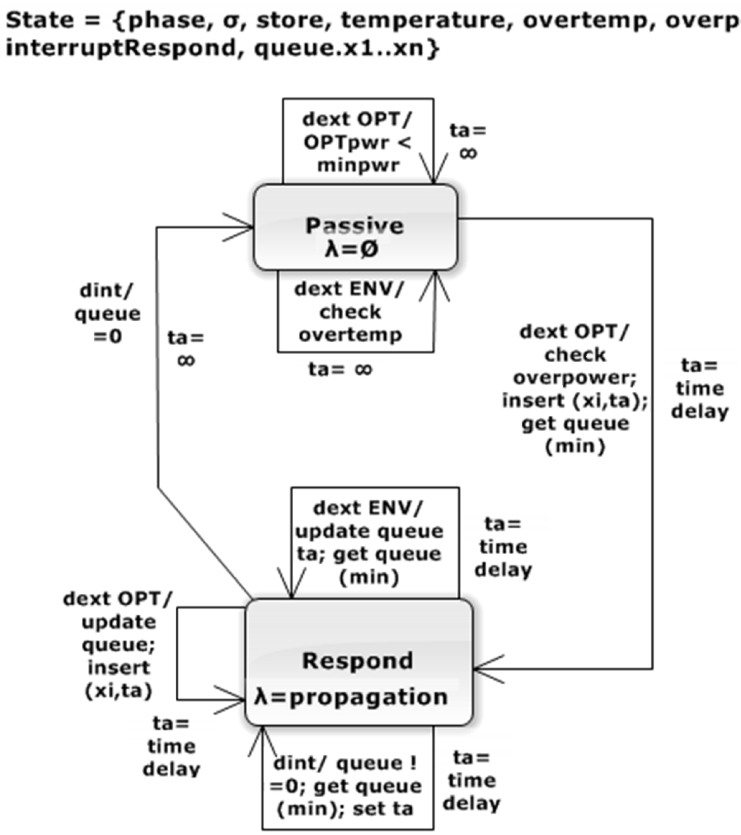

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
dext OPT/ OPTpwr < minpwr
ta= ∞
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dint/ queue =0
ta= ∞
dext OPT/ check overpower; insert (xi,ta); get queue (min)
ta= time delay
dext ENV/ update queue ta; get queue (min)
ta= time delay
dext OPT/ update queue; insert (xi,ta)
ta= time delay
Respond
λ=propagation
dint/ queue ! =0; get queue (min); set ta
ta= time delay

Figure 174. SM fiber phase transition diagram.

R.6 Event-Trace Diagram

This section shows various examples of packets entering the SM fiber. The tables list the states the SM fiber proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the SM fiber, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state. Note for optical fiber, the time for the respond phase is variable depending on the length of the fiber. In the following cases, the time of propagation through the fiber is set to 5.
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

R.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 70. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (x*i*, *tp*)**	**Notes: assume tp=5**
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s1	entry	respond	5	x1	c	n	n	n	null	
5	s1	exit	respond	inf	x1	c	n	n	n	null	
5	S2	entry	passive	inf	x1	c	n	n	n	null	

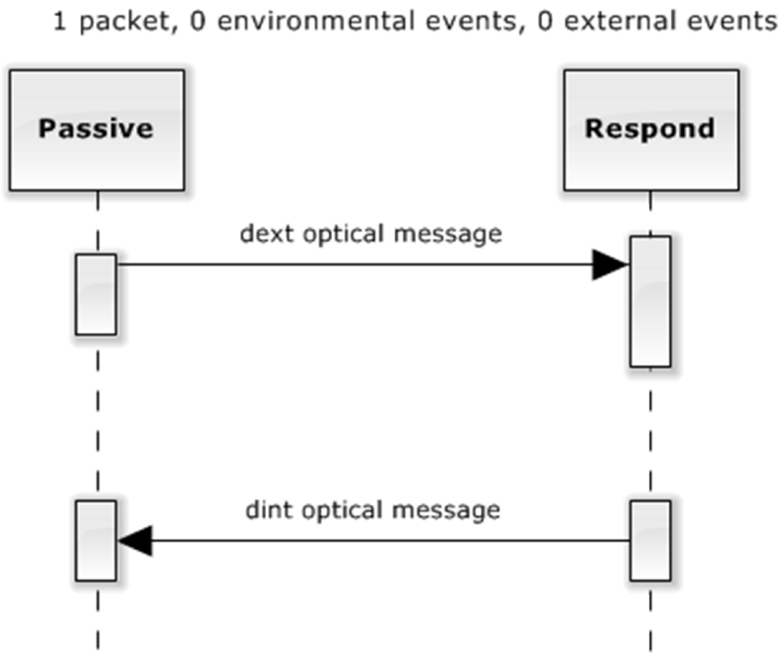


Figure 175. Case I sequence diagram.

R.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 71. *Case II state list.*

	state	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s1	entry	respond	5	x1	c	n	n	n	null	
2	s1	exit	respond	3	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s2	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s2	exit	respond	2	x2	c	n	n	n	null	
5	s3	entry	respond	2	x2	c	n	n	n	null	
7	s3	exit	respond	inf	x2	c	n	n	n	null	
7	s4	entry	passive	inf	x2	c	n	n	n	null	

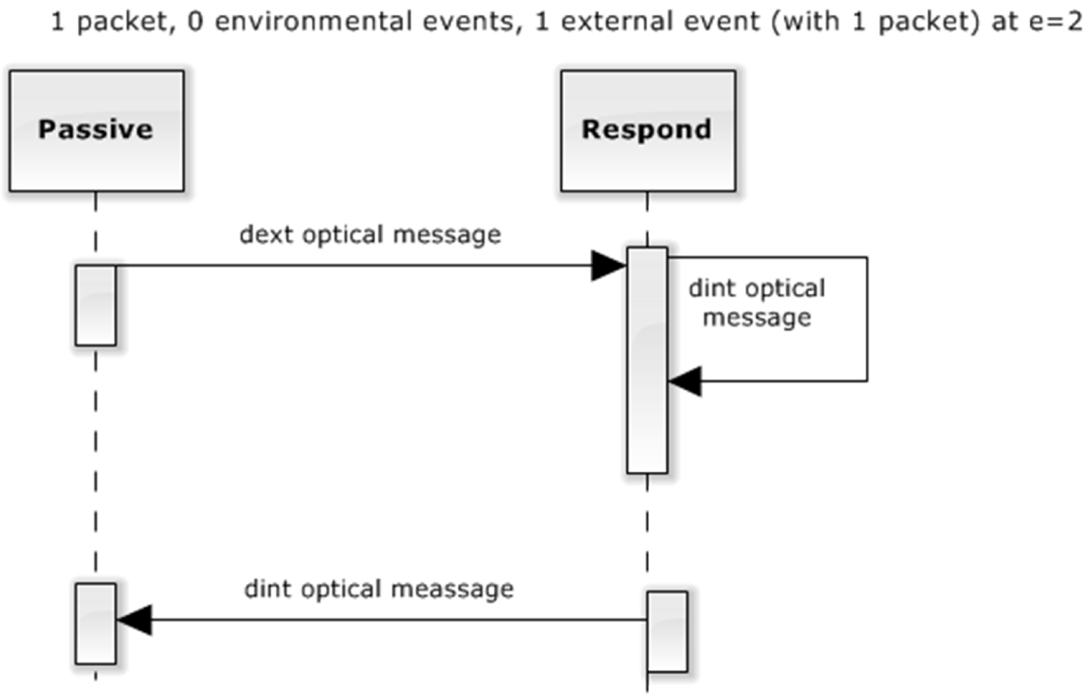


Figure 176. Case II sequence diagram.

R.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 72. *Case III state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s1	entry	respond	5	x1	c	n	n	n	null	
2	s1	exit	respond	3	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s2	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s2	exit	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x 4,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s3	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x	

										4,5)	
5	s3	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s4	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s4	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s5	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s5	exit	respond	0	x4	c	n	n	n	null	
8	s6	entry	respond	0	x4	c	n	n	n	null	
8	s6	exit	respond	inf	x4	c	n	n	n	null	
8	s7	entry	passive	inf	x4	c	n	n	n	null	

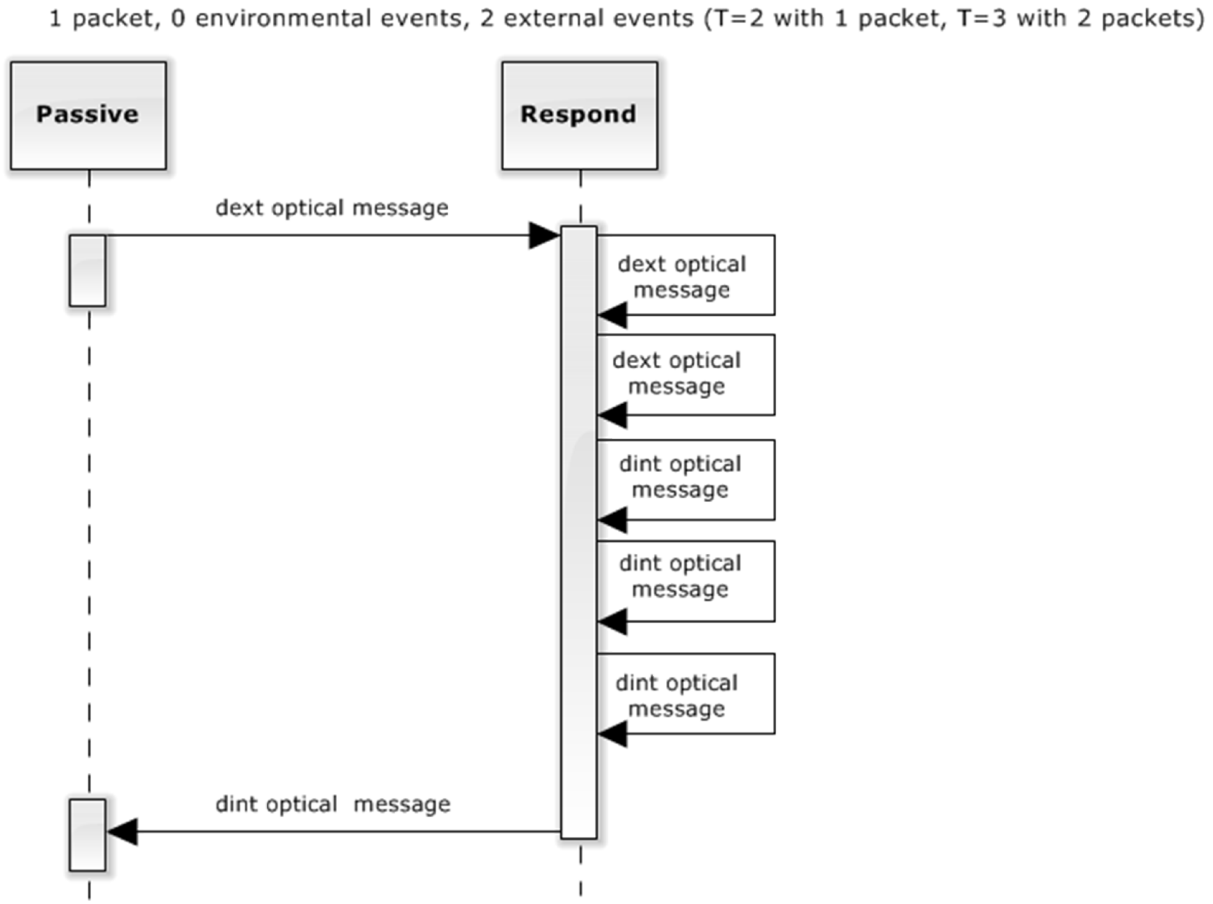


Figure 177. Case III sequence diagram.

R.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 73. *Case IV state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	y	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

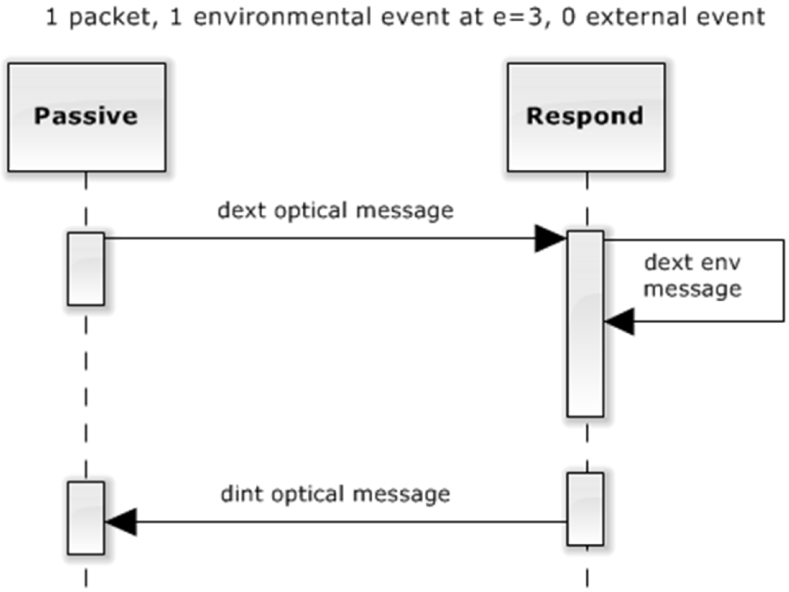


Figure 178. Case IV sequence diagram.

R.7 Single Mode Fiber Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.

- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
Peak power = full width, half maximum power calculation of the pulse

For the fixed SM fiber we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{SM fiber}$* **= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator

phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time delay$_i$) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay$_i$) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAttenDelay() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“OptIn_1”, “OptIn_2”, “EnvIn”} with
X_M = {(“OptIn_1”, V_{opt}), (“OptIn_2”, V_{opt}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“OptOut_1”, “OptOut_2”} with
Y_M = {(“OptOut_1”, $Y_{OptOut1}$), (“OptOut_2”, $Y_{OptOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower* *interruptRespond,* *queue*} = {{“passive”, “reflect”, “respond”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x {“Y”,”N”} x *V*}

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue, e, (($p_i$,$v_i$),….($p_n$,$v_n$))) =
 (“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                   queue.x1..xn)
   if phase = “passive” and p ∈ {“$\text{OptIn}_1$”, “$\text{OptIn}_2$”}
     for messagebag != null
        current = messagebag_first()
         if current.value.power > damaged.power
           overpower = “Y”
        if calcAtten(current.v) > MIN_POWER
         insert_event_q(current)
         remove_event_m(current)
     size= queue_size()
     if size > 0
       current = queue_min()
       time.delay = current.time.delay
       if InPort = “$\text{OptIn}_1$”
         outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
         outputPort = “$\text{OptOut}_2$”
```

```
      if InPort = “OptIn$_2$”
       outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
       outputPort = “OptOut$_1$”
     interruptRespond = “N”

 (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  if phase = “passive” and p $\in$ {“OptIn$_1$”, “OptIn$_2$”}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower =  “Y”
      if calcAtten(current.v) > MIN_POWER
       insert_event_q(current)
       remove_event_m(current)
    size= queue_size()
   if size < 0

(“respond”,  time.delay,  store,  temperature,  overtemp,  overpower,  interruptRespond,
                                                                    queue.x1..xn)
  if phase = “respond” and p $\in$ {“OptIn$_1$”, “OptIn$_2$”}
    update_delay(queue)
    for messagebag != null
     current = messagebag_first()
     if current.value.power > damaged.power
       overpower =  “Y”
     if calcAtten(current.v) > MIN_POWER
      insert_event_q(current)
      remove_event_m(current)
   interruptRespond= “Y”
   timeLeftRespond = timeLeftRespond - e

 (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = “passive” and p = “EnvIn”
   temperature = messagebag.temperature
   if temperature > damage.temp
     overtemp = “Y”

(“respond”,  time.delay,   store,  temperature,  overtemp,  overpower,  interruptRespond,
                                                                    queue.x1..xn)
   if phase = “respond” and p = “EnvIn”
    update_delay(queue)
    timeLeftRespond = time.delay- e
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”
    time.delay = timeLeftRespond
```

```
(phase, $\sigma - e$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
 ("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "respond" and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time_delay = current.time.delay
      if InPort = "$\text{OptIn}_1$"
        outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "$\text{OptOut}_2$"
      if InPort = "$\text{OptIn}_2$"
        outputPulse = calcAttenDelay(current.v, temperature, overtemp, peakpwr, overpwr)
        outputPort = "$\text{OptOut}_1$"
      interruptRespond= "N"

 ("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "respond" and size  = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

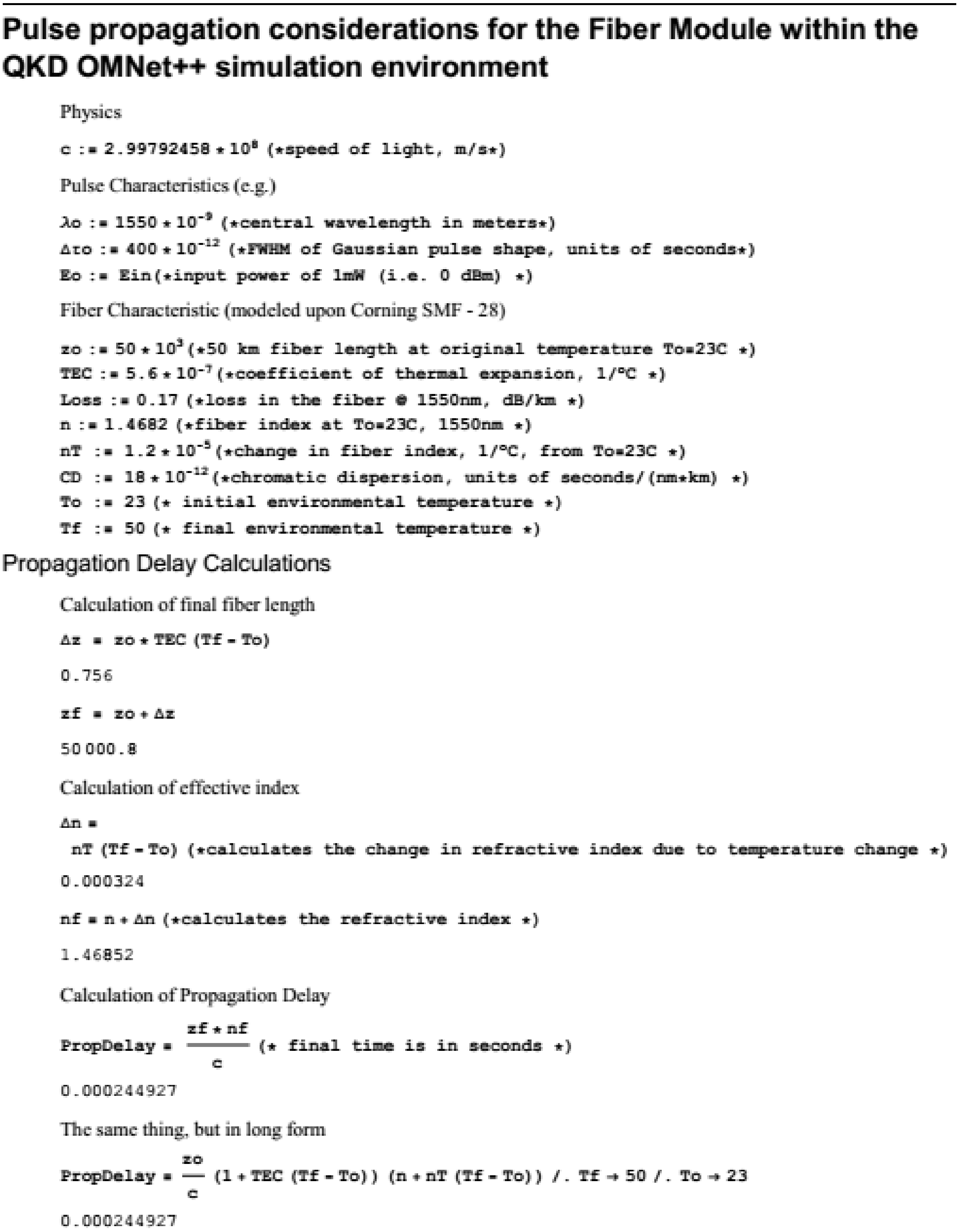
Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue) =
  (output.port, output.pulse)
      if phase = "respond"

  $\emptyset$ (null output)
     otherwise;
```

Time advance Function:

ta(*phase*, σ, *store, temperature*, *overtemp*, *overpower, interruptRespond, queue*) = σ;

R.8 Mathematical Model



Pulse propagation considerations for the Fiber Module within the QKD OMNet++ simulation environment

Physics

c := 2.99792458 * 10^{8} (*speed of light, m/s*)

Pulse Characteristics (e.g.)

λo := 1550 * 10^{-9} (*central wavelength in meters*)
Δτo := 400 * 10^{-12} (*FWHM of Gaussian pulse shape, units of seconds*)
Eo := Ein(*input power of 1mW (i.e. 0 dBm) *)

Fiber Characteristic (modeled upon Corning SMF - 28)

zo := 50 * 10^{3}(*50 km fiber length at original temperature To=23C *)
TEC := 5.6 * 10^{-7}(*coefficient of thermal expansion, 1/°C *)
Loss := 0.17 (*loss in the fiber @ 1550nm, dB/km *)
n := 1.4682 (*fiber index at To=23C, 1550nm *)
nT := 1.2 * 10^{-5}(*change in fiber index, 1/°C, from To=23C *)
CD := 18 * 10^{-12}(*chromatic dispersion, units of seconds/(nm*km) *)
To := 23 (* initial environmental temperature *)
Tf := 50 (* final environmental temperature *)

Propagation Delay Calculations

Calculation of final fiber length

Δz = zo * TEC (Tf - To)

0.756

zf = zo + Δz

50 000.8

Calculation of effective index

Δn =
 nT (Tf - To) (*calculates the change in refractive index due to temperature change *)

0.000324

nf = n + Δn (*calculates the refractive index *)

1.46852

Calculation of Propagation Delay

PropDelay = $\frac{zf * nf}{c}$ (* final time is in seconds *)

0.000244927

The same thing, but in long form

PropDelay = $\frac{zo}{c}$ (1 + TEC (Tf - To)) (n + nT (Tf - To)) /. Tf → 50 /. To → 23

0.000244927

Change in propagation delay through the fiber due to temperature

$$\mathtt{\Delta t = PropDelay - \frac{zo * n}{c}}$$

5.77406×10^{-8}

Chromatic Dispersion (pulse broadening) Calculations

For this example we assume the time-power profile of the pulse is a bandwidth-limited "ideal" Gaussian. Thus, the time-frequency bandwidth product is; Δτo*Δνo = 0.441. We need to calculate the spread in wavelength in the pulse to calcuate the pulse broadening

Calculation of original spectral (frequency) spread

$$\mathtt{\Delta\nu o = 0.441 / \Delta\tau o \;(*calculates\ the\ frequency\ FWHM\ for\ the\ original\ pulse,\ units\ of\ Hertz*)}$$

1.1025×10^{9}

$$\mathtt{\nu o = c / \lambda o \;//\; N \;(*calculates\ center\ frequency\ given\ central\ wavelength\ \lambda o,\ units\ of\ Hertz*)}$$

1.93414×10^{14}

$$\mathtt{\Delta\lambda = \frac{\Delta\nu o * (\lambda o)^{2}}{c} \;(*calculates\ the\ spectral\ FWHM,\ units\ in\ meters*)}$$

8.8353×10^{-12}

As can be seen, $\Delta\lambda$ is very small, so we should expect little pulse broadening due to chromatic dispersion.

$$\mathtt{\Delta\tau CD = CD * \frac{\Delta\lambda}{10^{-9}} * \frac{zf}{1000} \;(*spread\ due\ to\ chromatic\ dispersion,\ units\ of\ seconds\ *)}$$

7.95189×10^{-12}

$$\mathtt{\Delta\tau f = \sqrt{(\Delta\tau o)^{2} + (\Delta\tau CD)^{2}} \;(*final\ pulse\ duration,\ units\ of\ seconds\ *)}$$

4.00079×10^{-10}

The same thing, but in long form

$$\mathtt{\Delta\tau f2 = \sqrt{\left((\Delta\tau o)^{2} + \left(CD * \frac{1}{10^{-9}}\left(\frac{0.441 / \Delta\tau o * (\lambda o)^{2}}{c}\right) * \frac{zf}{1000}\right)^{2}\right)}}$$

4.00079×10^{-10}

Loss (final power) Calculations

Here we calculate the power of the pulse after it has passed through the full length of the fiber.

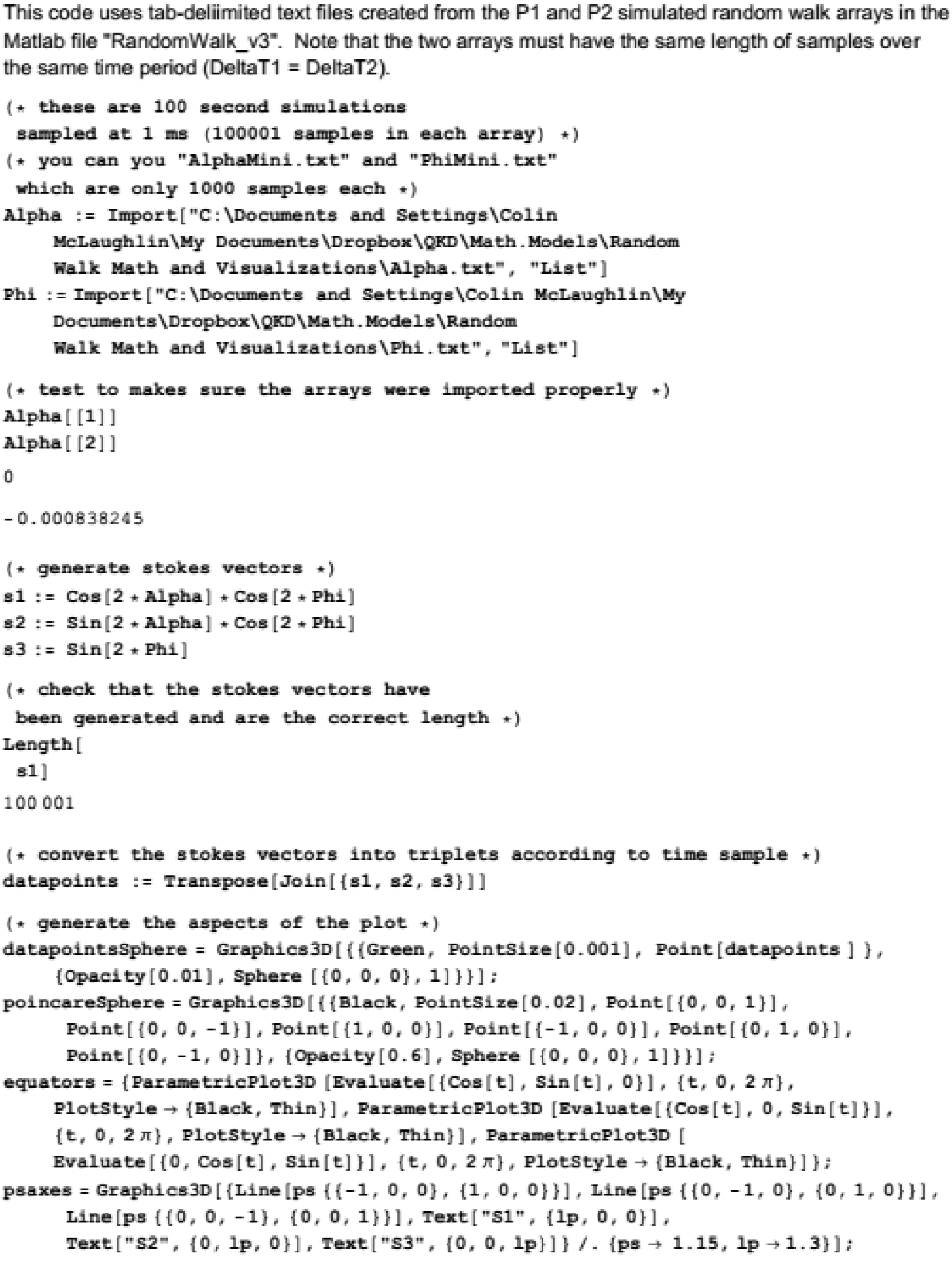
$$\mathtt{TotAtten = Loss * \frac{zf}{1000} \;(*"Total\ Attentuation"\ in\ units\ of\ dB\ *)}$$

8.50013

$$\mathtt{Ef = Eo * \sqrt{10^{-TotAtten/10}} \;(*\ output\ field\ *)}$$

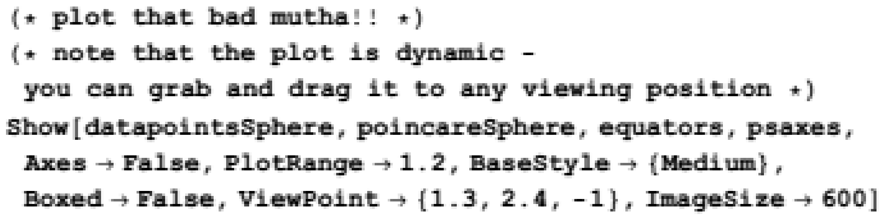
0.375832 Ein

Values used for output pulse

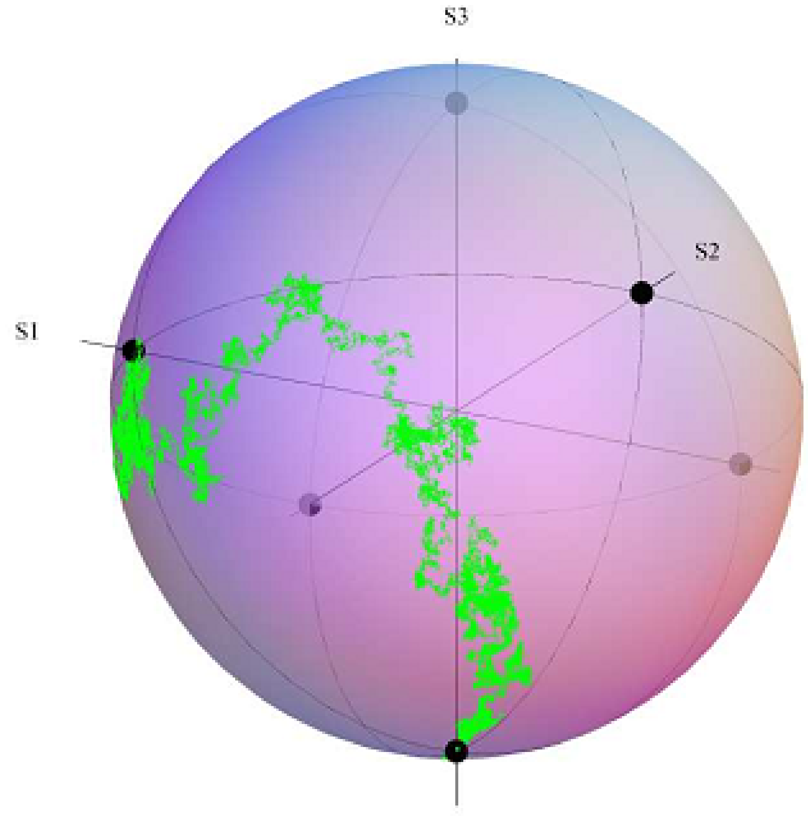


This code uses tab-deliimited text files created from the P1 and P2 simulated random walk arrays in the Matlab file "RandomWalk_v3". Note that the two arrays must have the same length of samples over the same time period (DeltaT1 = DeltaT2).

```
(* these are 100 second simulations
 sampled at 1 ms (100001 samples in each array) *)
(* you can you "AlphaMini.txt" and "PhiMini.txt"
 which are only 1000 samples each *)
Alpha := Import["C:\Documents and Settings\Colin
     McLaughlin\My Documents\Dropbox\QKD\Math.Models\Random
     Walk Math and Visualizations\Alpha.txt", "List"]
Phi := Import["C:\Documents and Settings\Colin McLaughlin\My
     Documents\Dropbox\QKD\Math.Models\Random
     Walk Math and Visualizations\Phi.txt", "List"]

(* test to makes sure the arrays were imported properly *)
Alpha[[1]]
Alpha[[2]]

0

-0.000838245

(* generate stokes vectors *)
s1 := Cos[2 * Alpha] * Cos[2 * Phi]
s2 := Sin[2 * Alpha] * Cos[2 * Phi]
s3 := Sin[2 * Phi]

(* check that the stokes vectors have
 been generated and are the correct length *)
Length[
 s1]

100 001

(* convert the stokes vectors into triplets according to time sample *)
datapoints := Transpose[Join[{s1, s2, s3}]]

(* generate the aspects of the plot *)
datapointsSphere = Graphics3D[{{Green, PointSize[0.001], Point[datapoints ] },
     {Opacity[0.01], Sphere [{0, 0, 0}, 1]}}];
poincareSphere = Graphics3D[{{Black, PointSize[0.02], Point[{0, 0, 1}],
      Point[{0, 0, -1}], Point[{1, 0, 0}], Point[{-1, 0, 0}], Point[{0, 1, 0}],
      Point[{0, -1, 0}]}, {Opacity[0.6], Sphere [{0, 0, 0}, 1]}}];
equators = {ParametricPlot3D [Evaluate[{Cos[t], Sin[t], 0}], {t, 0, 2 π},
     PlotStyle → {Black, Thin}], ParametricPlot3D [Evaluate[{Cos[t], 0, Sin[t]}],
     {t, 0, 2 π}, PlotStyle → {Black, Thin}], ParametricPlot3D [
     Evaluate[{0, Cos[t], Sin[t]}], {t, 0, 2 π}, PlotStyle → {Black, Thin}]};
psaxes = Graphics3D[{Line[ps {{-1, 0, 0}, {1, 0, 0}}], Line[ps {{0, -1, 0}, {0, 1, 0}}],
      Line[ps {{0, 0, -1}, {0, 0, 1}}], Text["S1", {lp, 0, 0}],
      Text["S2", {0, lp, 0}], Text["S3", {0, 0, lp}]} /. {ps → 1.15, lp → 1.3}];
```



```
(* plot that bad mutha!! *)
(* note that the plot is dynamic -
 you can grab and drag it to any viewing position *)
Show[datapointsSphere, poincareSphere, equators, psaxes,
 Axes → False, PlotRange → 1.2, BaseStyle → {Medium},
 Boxed → False, ViewPoint → {1.3, 2.4, -1}, ImageSize → 600]
```



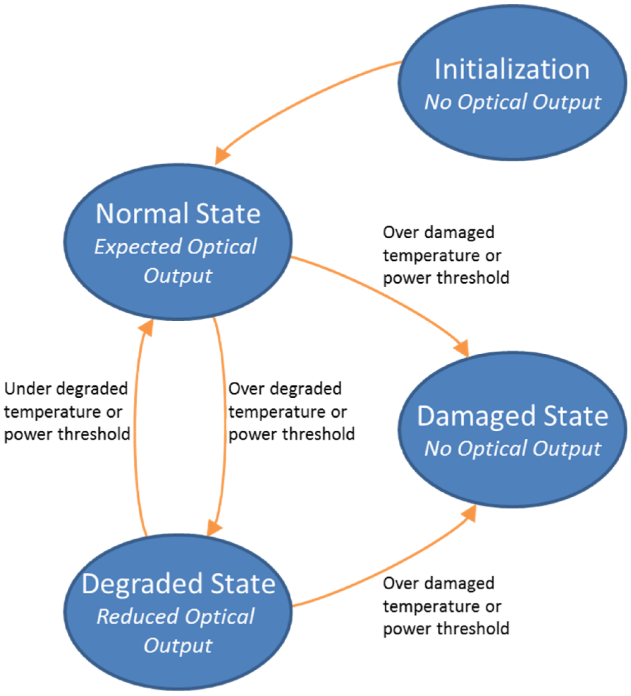
R.9 Component Use Case

R.9.1 Respond to an Optical Packet in the Single Mode Fiber (SM Fiber)

Optical packet arrives at the SM fiber. Place the optical packet into the optical queue. Check to see if optical packet overpowers the SM fiber. Records overpower condition, if applicable. Remove the optical packet from the queue and calculate the attenuated optical output signal based on the input signal, length and type of fiber, and the current component state. Propagate the attenuated optical output signal out of the component optical port that is not the same as the input port.

- Identified Alternative Uses Cases
    - React to an environmental message

- Assumptions
    - Component has completed initialization sequence at least once
    - Reflections are not affected by component state
    - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 179. Component states.

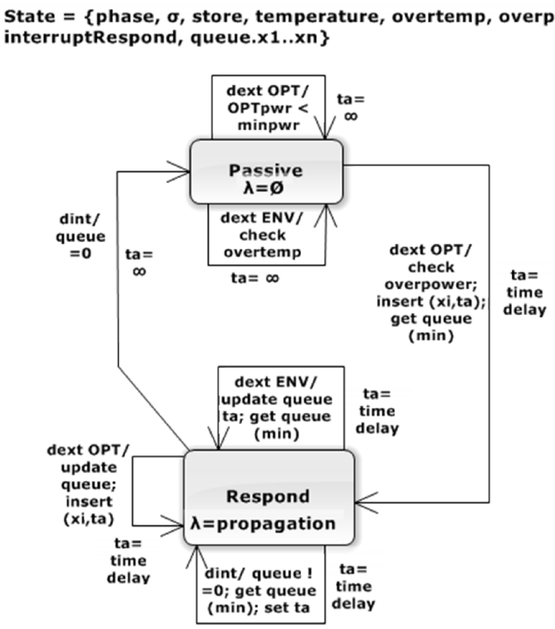

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
dext OPT/ OPTpwr < minpwr
ta= ∞
Passive λ=Ø
dint/ queue =0
ta= ∞
dext ENV/ check overtemp
ta= ∞
dext OPT/ check overpower; insert (xi,ta); get queue (min)
ta= time delay
dext ENV/ update queue ta; get queue (min)
ta= time delay
dext OPT/ update queue; insert (xi,ta)
ta= time delay
Respond λ=propagation
dint/ queue ! =0; get queue (min); set ta
ta= time delay

Figure 180. SM fiber phase transition diagram.

R.9.2 Respond to Optical Packet End Goals

- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Optical packet attenuated properly to the limit of accuracy
- Optical packet propagated out the correct port at the correct time

R.9.3 Respond to an Environmental Packet in the Single Mode Fiber (SM Fiber)

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
  - None

R.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

R.10 SM Fiber Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *SM Fiber Test Cases*

		Inject Ports				Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Env**	**Notes**	**opt #**	**env #**
Passive	1	1	0	0	single	1	0
	2	0	1	0	single	2	0
	3	0	0	1	single	2	1
	4	1	1	0	same time	4	1
	5	1	1	0	differ time	6	1
	6	1	1	1	same time	8	2
	7	1	1	1	differ time	10	3
	8	0	1	1	same time	11	4
	9	0	1	1	differ time	12	5
	10	1	0	1	same time	13	6
	11	1	0	1	differ time	14	7
	20	2	0	0	same time	16	7
	21	0	2	0	same time	18	7
	22	2	2	0	same time	22	7
	23	2	2	0	differ time	26	7
	24	2	2	1	same time	30	8
	25	2	2	1	differ time	34	9
	26	0	2	1	same time	36	10
	27	0	2	1	differ time	38	11
	28	2	0	1	same time	40	12
	29	2	0	1	differ time	42	13
totals		21	21	13	42		
Respond	41	2	0	0	single	44	13
	42	0	2	0	single	46	13
	43	1	0	1	single	47	14
	44	2	1	0	same time	50	14
	45	2	1	0	differ time	53	14
	46	2	1	1	same time	56	15
	47	2	1	1	differ time	59	16
	48	0	2	1	same time	61	17
	49	0	2	1	differ time	63	18
	50	2	0	1	same time	65	19
	51	2	0	1	differ time	67	20
	60	3	0	0	same time	70	20
	61	0	3	0	same time	73	20
	62	3	2	0	same time	78	20
	63	3	2	0	differ time	83	20
	64	3	2	1	same time	88	21
	65	3	2	1	differ time	93	22
	66	0	3	1	same time	96	23

67	0	3	1	differ time	99	24
68	3	0	1	same time	102	25
69	3	0	1	differ time	105	26
totals	36	27	13	63		
TC1	1	0	2	single	106	28
TC2	1	0	2	single	107	30
TC3	1	0	2	single	108	32
TC4	1	0	2	single	109	34
TC5	1	0	2	single	110	36
TC6	1	0	2	single	111	38
TC7	1	0	2	single	112	40
totals	7	0	14	7		

Notes: 5 - under minimum power packet sent on OPT1; OPT1 & OPT2 - differ time - Passive
7 - under minimum power packet sent on OPT2; OPT1, OPT2, ENV - differ time - Passive

R.11 References

Newport. (2013). Singlemode fiber, 1310/1550nm, 0.13 NA, 9.3µm MFD, 125µm cladding. Retrieved October 7, 2013, from http://search.newport.com/?q=*&x2=sku&q2=F-SMF-28

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). Single mode fiber. Retrieved October 7, 2013, from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=949

Appendix S - Micro-Electromechanical Systems (MEMS) Optical Switch

S.1 Device Description:

The optical switch is used to route light between one input port and two or more input/output ports. These devices usually consist of an electrically movable mirror that tilts to direct the light to either input/output port. The device is non-latching, meaning that electrical power must be applied for the device to maintain a connection between two of the ports. An optical connection is maintained to one output port when the electrical power is "off". Typically, optical switches have control interfaces that allow them to be mounted on circuit boards or have some other type of control port (DiConFiberOptics, 2013).

The micro-electromechanical systems (MEMS) are miniaturized devices integrating sensing and actuation functions to control systems (Zangari, 2010), powered by electrostatic actuators, and built using the same processes used in fabricating microelectronics. Typically, optical MEMS switches have some type of moving mirror, prism or holographic grating to deflect the light beams. Typical switching speeds are between 10ms and 10µs depending on the type of switching system. While easily fabricated, the major limitation of these devices is their slow response time, but they offer the advantage of low insertion loss and loss crosstalk between channels (Saleh & Teich, 1991). See Figure 1 for an example of an optical switch.

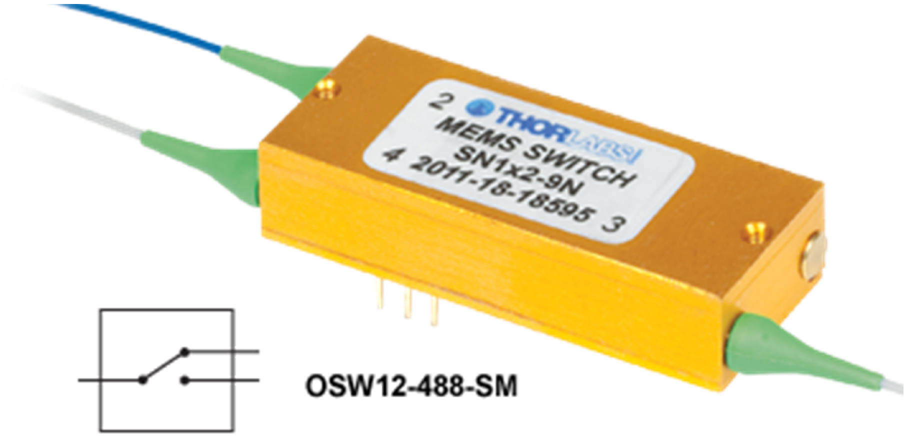


Figure 181. Example of an optical switch (ThorLabs, 2013).

The optical switch is a bidirectional optical component with three optical ports. Optical signals arriving at one of the ports is directed to one of the two input/output ports and propagated to the other port after a defined propagation delay. The switch is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the optical switch may become permanently damaged which changes its attenuation characteristics. Similarly, the optical switch is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the optical switch may become temporarily degraded or permanently damaged which changes its attenuation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the optical switch is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the optical switch. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the optical switch to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the optical switch using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the optical switch. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

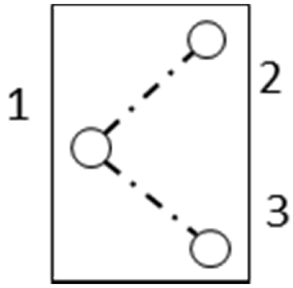


Figure 182. Symbol for the optical switch in the QKD system architecture.

S.2 Optical switch Conceptual Model

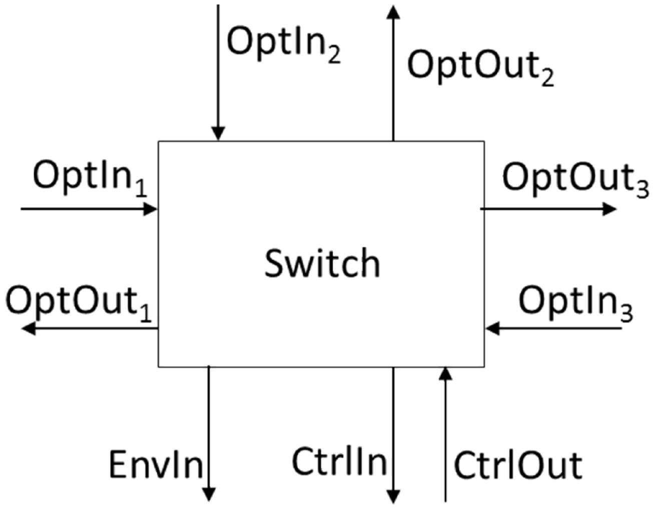


Figure 183. Optical switch conceptual model.

The conceptual model for an optical switch consists of three optical input ports {OptIn$_1$, OptIn$_2$, OptIn$_3$}, two optical output ports {OptOut$_1$, OptOut$_2$, OptOut$_3$}, one environmental input port {EvnIn} and one electrical controller input port and one electrical controller output port {CtrlIn, CtrlOut}. The environmental port allows external sources to communicate changes in the operational environment to the optical switch. The electrical controller ports allow for control inputs to the controller and responses from the optical switch to the higher system functions.

In comparison to the optical switch symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. The electrical control port is not shown for clarity in Figure 2, and is also decomposed in the model into an input port and an output port. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the optical switch, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 3 will instantaneously generate a reflected emitting out of OptOut_1.

The optical switch must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the optical switch can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

S.3 Mathematical Model

For a detailed mathematical description of the optical switch, refer to Section 17.8 which contains the Mathematica worksheet provided by the optical physics SME.

S.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the optical switch.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.

- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Place the optical packet into the queue
- Calculate the reflected power of the signal and send its output with the same port number.
- Retrieve the input optical signal from the queue and split it into two packets.
- Update the values for one packet as a ‘strong’ optical signal based on the characteristics of the switch, the original values of the input optical signal and the current environment and set the correct output port (the “normal signal”).
- Update the values for the other packet as a ‘weak’ optical signal based on the characteristics of the switch, the original values of the input optical signal and the current environment and set the correct output port (the “crosstalk” signal).
- Update the values of the input optical signal based on the characteristics of the switch, the original values of the input optical signal and the current environment.
- Send the changed output signal out of the optical output port numbers determined by the state of the switch and the crosstalk characteristics.

When an environmental message arrives:

- Update the CurrrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

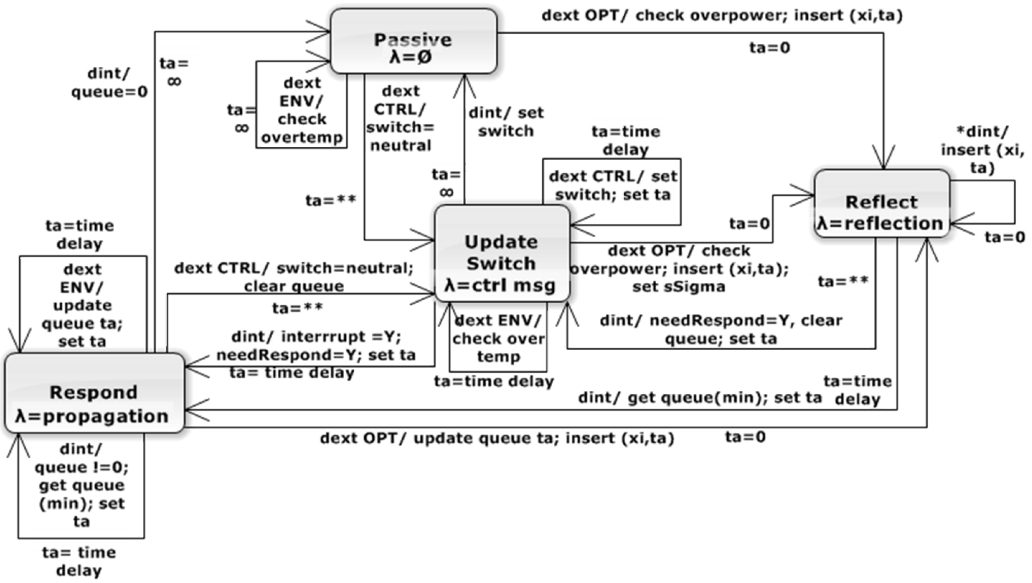
When a control message arrives:

- Change the optical path to the correct output port per the control message.
- Respond to the controller with an acknowledgement message.

S.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the optical switch in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the optical switch at the same time.

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, switchPosition, queue.x1..xn}



*** the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time**
**** ta = 0 or 15x10^9**

Figure 184. optical switch phase transition diagram.

S.6 Event-Trace Diagram

This section shows various examples of packets entering the optical switch. The tables list the states the optical switch proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the optical switch, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state
- Notes: any notes for that state

S.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 74. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**switch position**	**queue (x*i*, *tp*)**	**Notes: assume tp= 5**
	1-packet	no env	no ext	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	
0	s0	exit	passive	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	off	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	off	null	
5	s2	exit	respond	inf	x1	c	n	n	n	n	off	null	
5	s3	entry	passive	inf	x1	c	n	n	n	n	off	null	

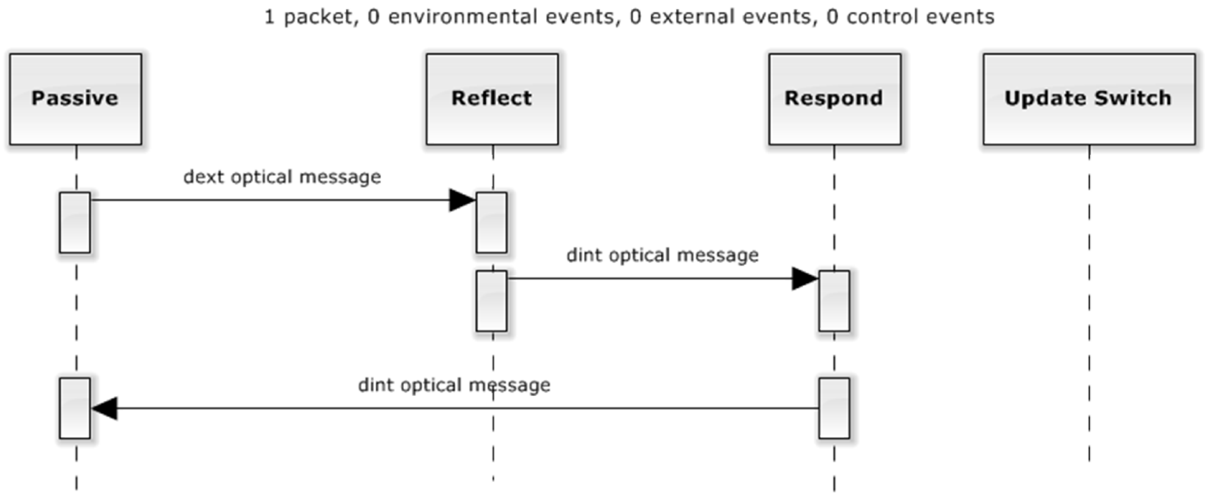


Figure 185. Case I sequence diagram.

S.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 75. *Case II state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**switch position**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	0 env	1 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	
0	s0	exit	passive	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	off	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	off	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	off	(x2,5)	dext at e=2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	off	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	off	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	off	(x2,5)	
5	s4	exit	respond	2	x2	c	n	n	n	n	off	null	
5	s5	entry	respond	2	x2	c	n	n	n	n	off	null	
7	s5	exit	respond	inf	x2	c	n	n	n	n	off	null	
7	s6	entry	passive	inf	x2	c	n	n	n	n	off	null	

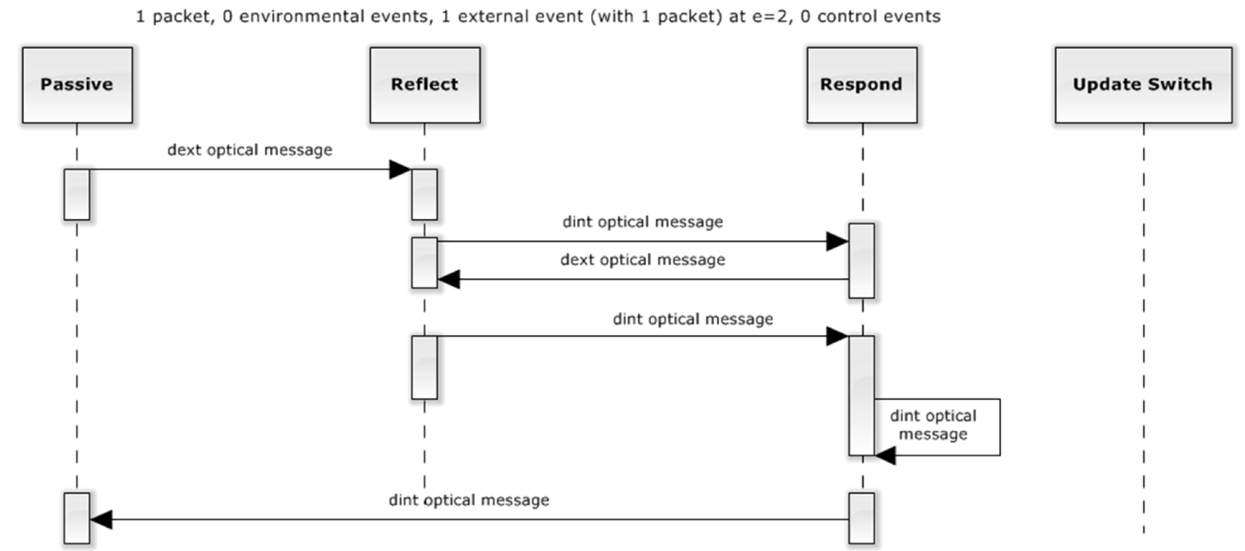


Figure 186. Case II sequence diagram.

S.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 76. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**switch position**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	0 env	2 opt	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	
0	s0	exit	passive	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	off	null	
0	s2	entry	respond	5	x1	c	n	n	n	n	off	null	
2	s2	exit	respond	0	x1	c	n	n	y	n	off	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	n	off	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	n	off	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	n	off	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	n	off	(x2,4) (x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	n	off	(x2,4) (x3,5)	

3	s5	exit	reflect	0	x1	c	n	n	y	n	off	(x2,4) (x3,5) (x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	n	off	(x2,4) (x3,5) (x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	n	off	(x2,4) (x3,5) (x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	n	off	(x2,4) (x3,5) (x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	n	off	(x3,2) (x4,2)	
5	s8	entry	respond	2	x2	c	n	n	n	n	off	(x3,2) (x4,2)	
7	s8	exit	respond	1	x3	c	n	n	n	n	off	(x4,0)	
7	s9	entry	respond	1	x3	c	n	n	n	n	off	(x4,0)	
8	s9	exit	respond	0	x4	c	n	n	n	n	off	null	
8	s10	entry	respond	0	x4	c	n	n	n	n	off	null	
8	s10	exit	respond	inf	x4	c	n	n	n	n	off	null	
8	s11	entry	passive	inf	x4	c	n	n	n	n	off	null	

1 packet, 0 environmental events, 2 external events (T=2 with 1 packet, T=3 with 2 packets), 0 control events

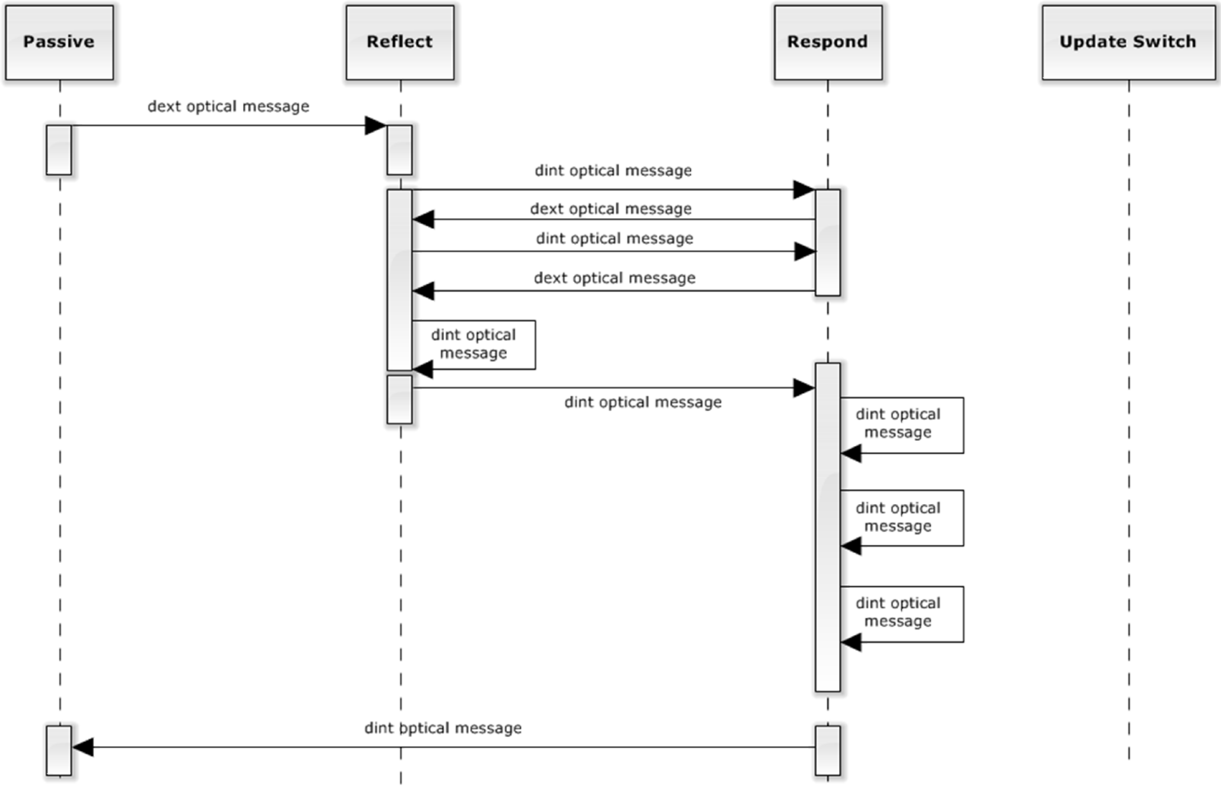


Figure 187. Case III sequence diagram.

S.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 77. *Case IV state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**switch position**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1-packet	1 env	0 ext	0 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	
0	s0	exit	passive	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	off	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	off	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	y	n	off	null	respond temp
3	s3	entry	respond	2	x1	c	y	n	y	n	off	null	
5	s3	exit	respond	inf	x1	c2	y	n	n	n	off	null	
5	s4	entry	passive	inf	x1	c2	y	n	n	n	off	null	

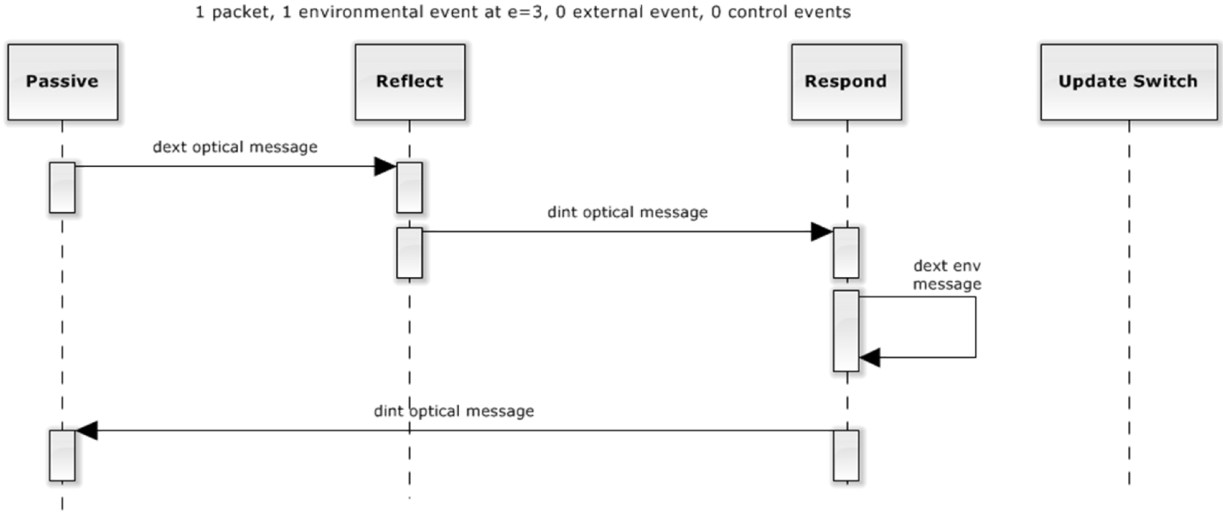


Figure 188. Case IV sequence diagram.

S.6.5 CASE V: Initial Passive with Single Optical Packet Arriving at Time 0, Single Control Packet Arriving at Time

Table 78. *Case V state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**need respond**	**switch position**	**queue (*xi*, *tp*)**	**Notes: assume tp= 5**
	1 opt	1 env	0 opt	1 ctrl									
0	s0	entry	passive	inf	null	c	n	n	n	n	off	null	
0	s0	exit	passive	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	n	off	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	n	off	(x1,5)	
0	s2	entry	respond	5	x1	c	n	n	n	n	off	(x1,5)	CTRL arrives e=3
3	s2	exit	respond	15x10^9	null	c	n	n	n	n	neutral	null	
3	s3	entry	update switch	15x10^9	null	c	n	n	n	n	neutral	null	
15x10^9	s3	exit	update switch	inf	null	c	n	n	n	n	on	null	
15x10^9	s4	entry	passive	inf	null	c	n	n	n	n	off	null	

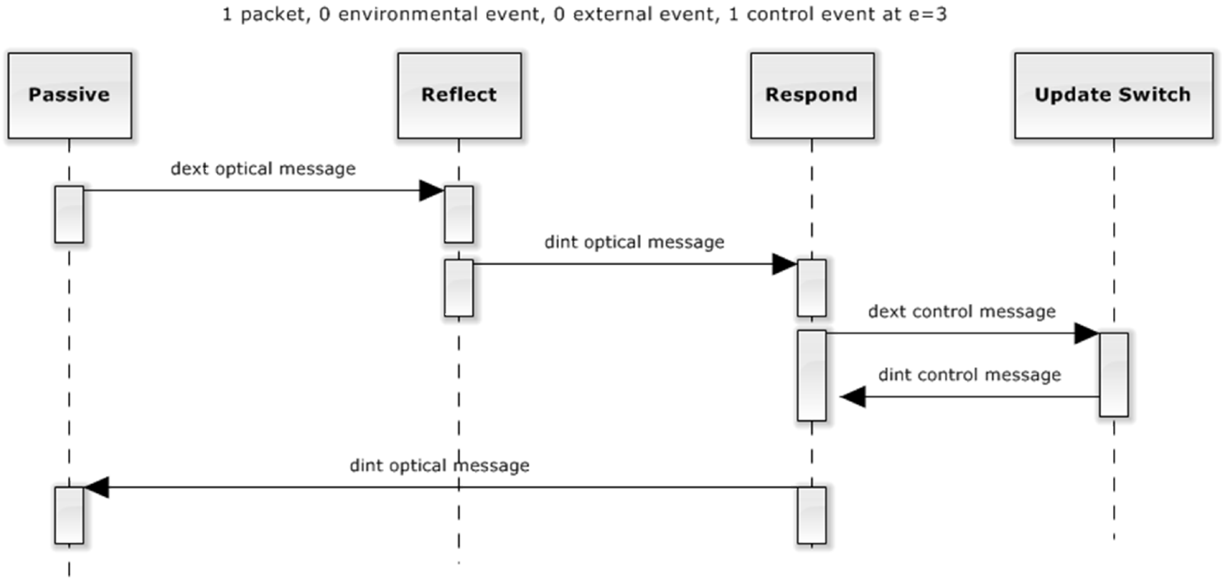


Figure 189. Case V sequence diagram.

S.7 Optical switch Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- Assume that only one control packet will arrive at any given time, due to the small time scales involved and the length of time necessary for optical path changes.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interrruptRespond”, “needRespond”, “switchPosition”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
Peak power = full width, half maximum power calculation of the pulse

For the optical switch we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

$\boldsymbol{DEVS_{optical\ switch} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$

where

t_p = transmission time inside the attenuator

temperature = current temperature of the attenuator

phase = control state that keeps track of the internal phase of the attenuator

phase = {“passive”, “reflect”, “respond”, “update detector”}

overtemp = flag variable set when device meets or exceeds damaged temperature level

overpower = flag variable set when device meets or exceeds damaged optical power level

interruptRespond = flag variable set when Respond phase is interrupted by an external event

needRespond= flag variable set when both Reflect and UpdateDetector respond to inputs

switchPosition= variable that holds the current switch position {“on”, “off”, “neutral”}

swtchSigma = variable that holds the time advance for the Update Switch phase

attenpower = variable the holds the attenuated power of the current optical packet

peak.power = variable the holds the peak power of the current optical packet

messagebag= variable that stores the current *x* input value(s) (*p*,*v)*

damaged.power = variable that holds the component damaged optical power level parameter

damage.temp = variable that holds the component damaged temperature level parameter

current = variable that stores the queue event being manipulated

need.reflect= variable that stores queue event that needs reflecting

reflect = variable that stores the current reflected optical packet

reflect.port = variable that holds the current reflection output port

reflect.power = variable that holds the current reflection power

time.delay = variable that stores the time delay in the queue for event *i*

output.pulse= variable that stores the output optical packet

output.port = variable that holds the output optical packet port

size= variable that holds the number of events in the queue

ctrlOutput = variable that stores the output control message response

queue.current = variable that holds the currently selected queue event

store = variable that holds values of the current optical packet

timeLeftRespond = time left in Respond phase for the current optical packet

e = elapsed time since last transition occurred

σ = state variable that holds the time to next transition

queue = input container object to store the scheduled inputs

queue_size() = method that returns number of entries in the queue

queue_min() = method that removes the queue entry with the smallest time delay

queue_first() = method that returns the first element of the queue

queue_need_reflected() = method returns the first unreflected queue event

empty_all_q() = method that empties the queue

messagebag_first() = method that returns the first element of the message bag

mark_reflected() = method that marks the current queue event as being reflected

update_delay() = method that updates the time delay of entries in the queue by *e*

ctrlMsg() = method that generates a response message to received control messages

outputMsg() = method that generates the response message to received optical packets

insert_event_q() = method that inserts the current (x$_i$, time delay$_i$) into the queue

remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay_i) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReflected() = method that calculates reflection power of an optical packet
changePolarization() = method that changes current polarization of the PM
calcAttenPolar() = method that calculates the optical output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr, currentPolarization*)
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
$q.v_{min}$ = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"OptIn_1", "OptIn_2", " OptIn_3", "EnvIn", "CtrlIn"} with
X_M = {("OptIn_1", V_{opt}), ("OptIn_2", V_{opt}), ("OptIn_3", V_{opt}), ("EnvIn", V_{env}), ("CtrlIn", V_{ctrl})} is the set of input ports and values.

OutPorts = {"OptOut_1", "OptOut_2", "OptOut_3", "CtrlOut"} with
Y_M = {("OptOut_1", $Y_{OptOut1}$), ("OptOut_2", $Y_{OptOut2}$), ("OptOut_3", $Y_{OptOut3}$), ("CtrlOut", $Y_{CtrlOut}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, interruptRespond, needRespond, switchPosition, queue*} = {{"passive", "reflect", "respond", "update polarization"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x {"Y","N"} x {"Y","N"} x {"off", "on", "neutral"} x *V*}

External Transition Function:

δ_{ext}(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond , switchPosition, queue*, *e*, ((p_i,v_i),…. (p_n,v_n))) =

```
(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                switchPosition, queue.x1..xn)
  if phase = “passive” and p ∈ {“$\text{OptIn}_1$”, “$\text{OptIn}_2$”, “$\text{OptIn}_3$”}
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower =  “Y”
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue.first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = “N”

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                switchPosition, queue.x1..xn)
  if phase = “respond” and p ∈ {“ $\text{OptIn}_1$”, “$\text{OptIn}_2$”, “$\text{OptIn}_3$”}
    update_delay(queue)
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower =  “Y”
      insert_event_q(current)
      remove_event_m(current)
   queue.current = queue_need_reflected()
   reflect = (queue.current.p), calcReflected(queue.current.v))
   mark_reflected(queue.current)
   interruptRespond= “Y”
   timeLeftRespond = timeLeftRespond - e

(“reflect”, 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                switchPosition, queue.x1..xn)
  if phase = “update switch” and p ∈ {“$\text{OptIn}_1$”, “$\text{OptIn}_2$”, “$\text{OptIn}_3$”}
    update_delay(queue)
    for messagebag != null
      current = messagebag_first()
      if current.value.power > damaged.power
        overpower =  “Y”
      insert_event_q(current)
      remove_event_m(current)
    queue.current = queue_first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    switchSigma = switchSigma– e
```

```
("passive", ∞, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                            switchPosition, queue.x1..xn)
   if phase = "passive" and p = "EnvIn"
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = "Y"

("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                needRespond, switchPosition, queue.x1..xn)
   if phase = "respond" and p = "EnvIn"
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"
     time.delay = timeLeftRespond

("update switch", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                needRespond, switchPosition, queue.x1..xn)
   if phase = "update switch" and p = "EnvIn"
     update_delay(queue)
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = "Y"
     switchSigma = switchSigma– e

("update switch", $15x10^9$, store, temperature, overtemp, overpower, interruptRespond,
                                                needRespond, switchPosition, queue.x1..xn)
  if phase = {"respond", "passive"} and switchPosition = 3 and p = "CtrlIn"
    if switchPosition = "off" and store.value.voltage = "on"
      switchPosition = "neutral"
      ctrlOutput = ctrlMsg(store)
      empty_all_q()
    if switchPosition = "on" and store.value.voltage = "off"
      switchPosition = "neutral"
      ctrlOutput = ctrlMsg(store)
      empty_all_q()

("update switch", 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                switchPosition, queue.x1..xn)
  if phase = "respond" and switchPosition != 3 and p = "CtrlIn"
     update_delay(queue)
     ctrlOutput = ctrlMsg(store)
     interruptRespond= "Y"
```

```
("update switch", 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                  switchPosition, queue.x1..xn)
  if phase = "passive" and switchPosition != 3 and p = "CtrlIn"
    ctrlOutput = ctrlMsg(store)


("update switch", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                  needRespond, switchPosition, queue.x1..xn)
  if phase = "update switch" and p = "CtrlIn"
    switchPosition = store.value.voltage
    switchSigma = switchSigma– e

(phase, $\sigma - e$, $\infty$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                  switchPosition, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                  switchPosition, queue)=

 ("reflect", 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                  switchPosition, queue.x1..xn))

  if phase = "reflect" and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
    reflect = (current.p), calcReflected(current.v))
    mark_reflected(current)

 ("update switch", $15x10^9$, store, temperature, overtemp, overpower, interruptRespond,
                                                  needRespond, switchPosition, queue.x1..xn)
  if phase = "reflect" and need.reflect = null and switchPosition =3
    need.reflect = queue_need_reflected()
    empty_all_q()

 ("update switch", 0, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                  switchPosition, queue.x1..xn)
    if phase = "reflect" and need.reflect = null and switchPosition < 3
      need.reflect = queue_need_reflected()
      ctrlOutput = ctrlMsg(store)

 ("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                                  needRespond, switchPosition, queue.x1..xn)
  if phase = "reflect" and need.reflect = null
    need.reflect = queue_need_reflected()
```

```
   if interruptRespond = “N”
     current = queue_min()
     time.delay = current.time.delay
     if InPort = “OptIn$_1$” and switchPosition = “off”
      new1 = (“OptOut$_2$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     if InPort = “OptIn$_1$” and switchPosition = “on”
      new1 = (“OptOut$_3$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_2$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     if InPort = “OptIn$_2$” and switchPosition = “off”
      new1 = (“OptOut$_1$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     if InPort = “OptIn$_3$” and switchPosition = “on”
      new1 = (“OptOut$_1$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
    timeLeftRespond = propagation delay
  else
    time.delay = timeLeftRespond

(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                      needRespond, switchPosition, queue.x1..xn)
  if phase = “respond” and size > 0
    update_delay(queue)
    size= queue_size()
    current = queue_min()
    time.delay = current.time.delay
     if InPort = “OptIn$_1$” and switchPosition = “off”
      new1 = (“OptOut$_2$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     if InPort = “OptIn$_1$” and switchPosition = “on”
      new1 = (“OptOut$_3$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_2$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     if InPort = “OptIn$_2$” and switchPosition = “off”
      new1 = (“OptOut$_1$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
     if InPort = “OptIn$_3$” and switchPosition = “on”
      new1 = (“OptOut$_1$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
      new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
   interruptRespond= “N”

 (“passive”, $\infty$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                switchPosition, queue.x1..xn)
  if phase = “respond” and size  = 0
    size= queue_size()
```

```
(“passive”, $\infty$, store, temperature, overtemp, overpower, overpower, interruptRespond,
                                        needRespond, switchPosition, queue.x1..xn)
  if phase = “update switch”  and switchPosition =3
    if store.value.voltage= “off” and switchPosition = 3
      switchPosition = “off”
    if store.value.voltage= “on” and switchPosition = 3
      switchPosition = “on”

(“passive”, $\infty$, store, temperature, overtemp, overpower, overpower, interruptRespond,
                                        needRespond, switchPosition, queue.x1..xn)
  if phase = “update switch”  and needRespond = “N”


(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
                                        needRespond, switchPosition, queue.x1..xn)
  if phase = “update switch” and interruptRespond = “N” and needRespond = “Y”
    current = queue_min()
    time.delay = current.time.delay
      if InPort = “OptIn$_1$” and switchPosition = “off”
        new1 = (“OptOut$_2$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
        new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
      if InPort = “OptIn$_1$” and switchPosition = “on”
        new1 = (“OptOut$_3$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
        new2 = (“OptOut$_2$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
      if InPort = “OptIn$_2$” and switchPosition = “off”
        new1 = (“OptOut$_1$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
        new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
      if InPort = “OptIn$_3$” and switchPosition = “on”
        new1 = (“OptOut$_1$”, calcStrong(current.v, temperature, overtemp, peakpwr, overpwr)
        new2 = (“OptOut$_3$”, calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, needRespond,
                                                        switchPosition, queue) =
  (reflect.p, reflect.v)
    if phase = “reflect”

  (new1.p, new1.v)
    if phase = “respond”

  (new2.p, new2.v)
    if phase = “respond”
```

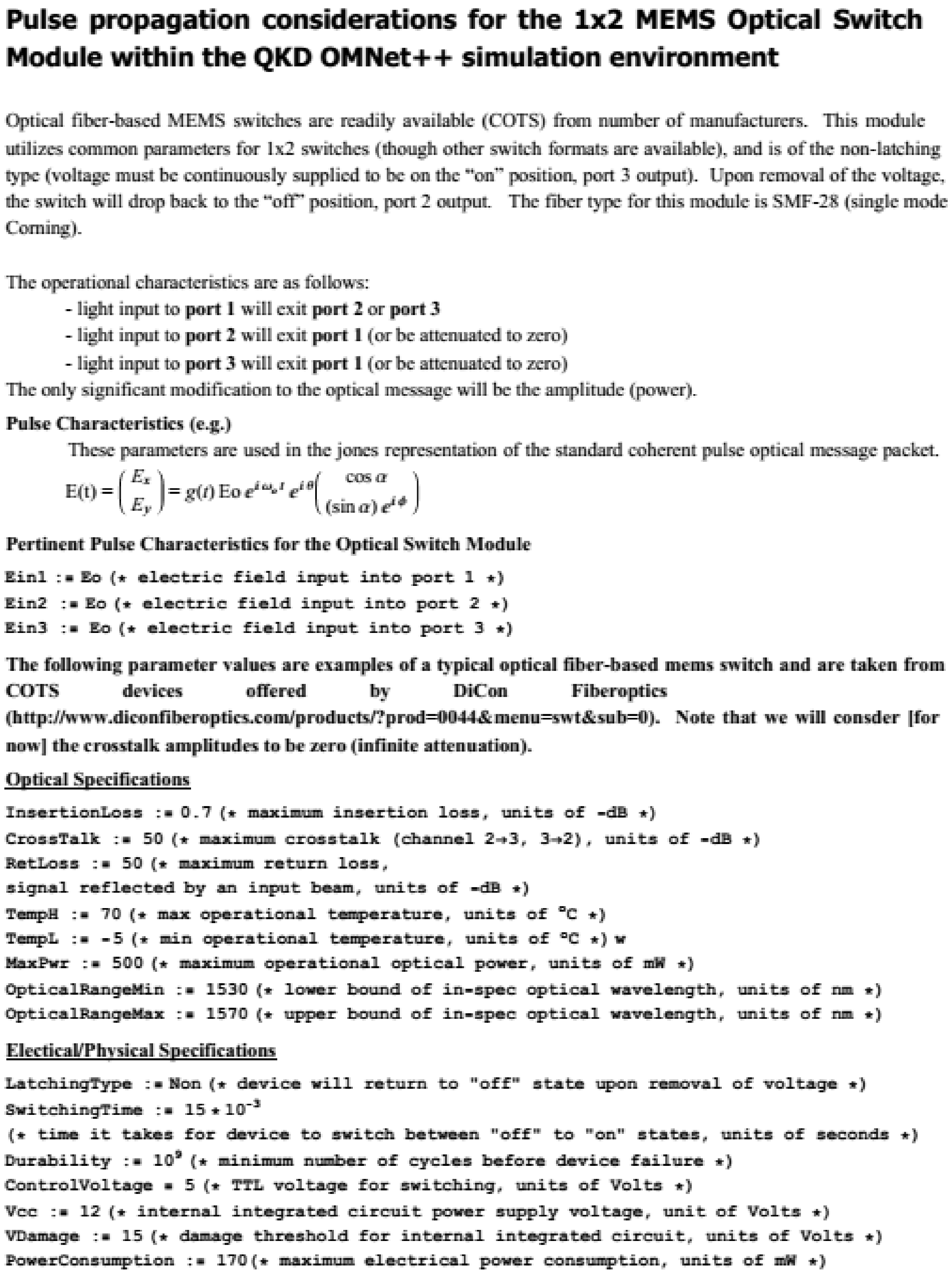
("CtrlOut", *ctrlOutput*)
  if phase = "update switch"

∅ (null output)
  otherwise;

Time advance Function:

ta(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, interruptRespond, needRespond, switchPosition, queue*) = *σ*;

S.8 Mathematical Model



Pulse propagation considerations for the 1x2 MEMS Optical Switch Module within the QKD OMNet++ simulation environment

Optical fiber-based MEMS switches are readily available (COTS) from number of manufacturers. This module utilizes common parameters for 1x2 switches (though other switch formats are available), and is of the non-latching type (voltage must be continuously supplied to be on the "on" position, port 3 output). Upon removal of the voltage, the switch will drop back to the "off" position, port 2 output. The fiber type for this module is SMF-28 (single mode Corning).

The operational characteristics are as follows:

- light input to **port 1** will exit **port 2** or **port 3**
- light input to **port 2** will exit **port 1** (or be attenuated to zero)
- light input to **port 3** will exit **port 1** (or be attenuated to zero)

The only significant modification to the optical message will be the amplitude (power).

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\,\omega_o t}\,e^{i\,\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Optical Switch Module

```
Ein1 := Eo (* electric field input into port 1 *)
Ein2 := Eo (* electric field input into port 2 *)
Ein3 := Eo (* electric field input into port 3 *)
```

The following parameter values are examples of a typical optical fiber-based mems switch and are taken from COTS devices offered by DiCon Fiberoptics (http://www.diconfiberoptics.com/products/?prod=0044&menu=swt&sub=0). Note that we will consder [for now] the crosstalk amplitudes to be zero (infinite attenuation).

Optical Specifications

```
InsertionLoss := 0.7 (* maximum insertion loss, units of -dB *)
CrossTalk := 50 (* maximum crosstalk (channel 2→3, 3→2), units of -dB *)
RetLoss := 50 (* maximum return loss,
signal reflected by an input beam, units of -dB *)
TempH := 70 (* max operational temperature, units of °C *)
TempL := -5 (* min operational temperature, units of °C *) w
MaxPwr := 500 (* maximum operational optical power, units of mW *)
OpticalRangeMin := 1530 (* lower bound of in-spec optical wavelength, units of nm *)
OpticalRangeMax := 1570 (* upper bound of in-spec optical wavelength, units of nm *)
```

Electical/Physical Specifications

```
LatchingType := Non (* device will return to "off" state upon removal of voltage *)
SwitchingTime := 15 * 10^-3
(* time it takes for device to switch between "off" to "on" states, units of seconds *)
Durability := 10^9 (* minimum number of cycles before device failure *)
ControlVoltage = 5 (* TTL voltage for switching, units of Volts *)
Vcc := 12 (* internal integrated circuit power supply voltage, unit of Volts *)
VDamage := 15 (* damage threshold for internal integrated circuit, units of Volts *)
PowerConsumption := 170(* maximum electrical power consumption, units of mW *)
```

Operational/Attenuation Calculations for 1x2 Optical Switch

The operation of this device will have three "states";
- State 1; "off" - optical power is throughput from **port 1** to **port 2** and vice-versa.
- State 2; "neutral" - optical ouput power is zero from all ports. This will last for "SwitchingTime" duration, and will occur after recieving a TTL on (+5 V) or TTL off (0 V) control message
- State 3; "on" - optical power is throughput from **port 1** to **port 3** and vice-versa.

State 1;

`Eout2[Ein1_, InsertionLoss_] := Ein1 *` $\sqrt{10^{-\text{InsertionLoss}/10}}$
`(* case: optical input on port 1 *)`
`Eout3[Ein1_, InsertionLoss_] := Ein1 * 0 (* case: optical input on port 1 *)`

`Eout1[Ein2_, InsertionLoss_] := Ein2 *` $\sqrt{10^{-\text{InsertionLoss}/10}}$
`(* case: optical input on port 2 *)`
`Eout1[Ein3_, InsertionLoss_] := Ein3 * 0 (* case: optical input on port 3 *)`

State 2;

```
Eout2[Ein1_, InsertionLoss_] := Ein1 * 0 (* case: optical input on port 1 *)
Eout3[Ein1_, InsertionLoss_] := Ein1 * 0 (* case: optical input on port 1 *)
Eout1[Ein2_, InsertionLoss_] := Ein2 * 0 (* case: optical input on port 2 *)
Eout1[Ein3_, InsertionLoss_] := Ein3 * 0 (* case: optical input on port 3 *)
```

State 3;

`Eout2[Ein1_, InsertionLoss_] := Ein1 * 0 (* case: opitcal input on port 1 *)`

`Eout3[Ein1_, InsertionLoss_] := Ein1 *` $\sqrt{10^{-\text{InsertionLoss}/10}}$
`(* case: optical input on port 1 *)`
`Eout1[Ein2_, InsertionLoss_] := Ein2 * 0 (* case: optical input on port 2 *)`
`Eout1[Ein3_, InsertionLoss_] :=`
`Ein3 *` $\sqrt{10^{-\text{InsertionLoss}/10}}$ `(* case: optical input on port 3 *)`

If we wish to flag the optical switch to include **undesired return (reflected)** messages, the following operations would hold true,

`Eout1[Ein1_, RetLoss_] := Ein1 *` $\sqrt{10^{-\text{RetLoss}/10}}$

`Eout2[Ein2_, RetLoss_] := Ein2 *` $\sqrt{10^{-\text{RetLoss}/10}}$

`Eout3[Ein3_, RetLoss_] := Ein3 *` $\sqrt{10^{-\text{RetLoss}/10}}$

Polarizaion Calculations for 1x2 Optical Switch

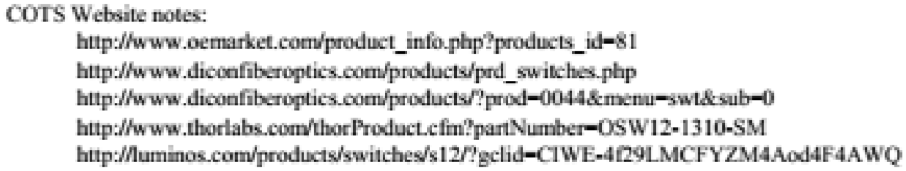
For a the 1x2 optical switch, the polariztion change can be related to that of a similar length of fiber, in this case SMF-28. This will be incorporated into the system polarization state randomization and drift, and so will not be included in this module. Using the state assignments from the above Operational/Attenuation calculations,

State 1;

```
OutputPol2[InputPol1_] := InputPol1
OutputPol1[InputPol2_] := InputPol2
```

State 3;

```
OutputPol3[InputPol1_] := InputPol1
OutputPol1[InputPol3_] := InputPol3
```



COTS Website notes:

http://www.oemarket.com/product_info.php?products_id=81
http://www.diconfiberoptics.com/products/prd_switches.php
http://www.diconfiberoptics.com/products/?prod=0044&menu=swt&sub=0
http://www.thorlabs.com/thorProduct.cfm?partNumber=OSW12-1310-SM
http://luminos.com/products/switches/s12/?gclid=CIWE-4f29LMCFYZM4Aod4F4AWQ

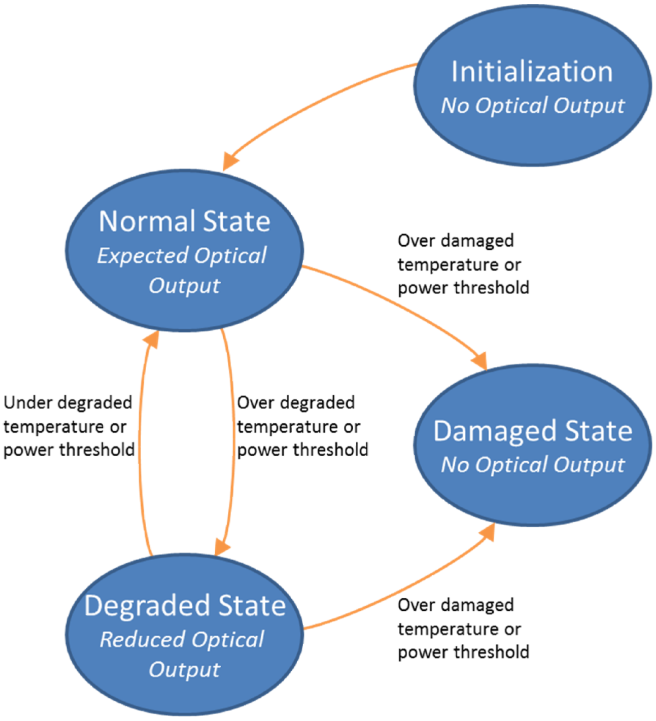
S.9 Component Use Case

S.9.1 Respond to an Optical Packet in the Optical Switch

Optical packet arrives at the optical switch. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the optical switch. Records overpower condition, if applicable. Remove the optical packet from the queue and create transmitted and crosstalk packets. Calculate the attenuated optical output signals based on the input signal and the current component state. Propagate the attenuated optical output signals out of the component optical ports based on the input port and switch position.

- Identified Alternative Uses Cases
  - Respond to a control message
  - React to an environmental message

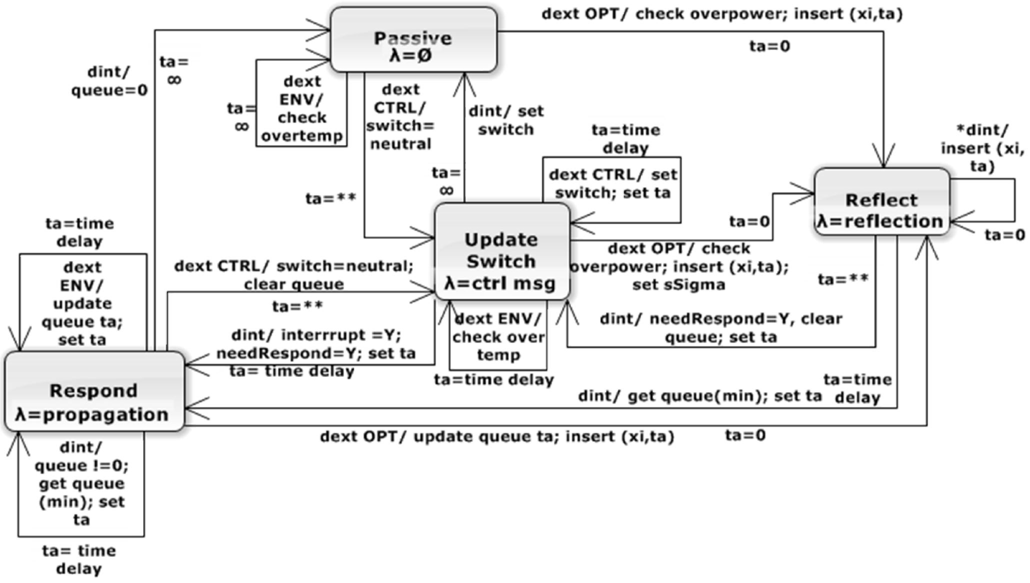
- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Damaged State
No Optical Output
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 190. Component states.

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, needRespond, switchPosition, queue.x1..xn}



* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time
** ta = 0 or 15x10^9

Figure 191. optical switch phase transition diagram.

S.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Optical packet entered and removed from queue in proper sequence
- Overpower condition properly recognized and recorded
- Optical packet attenuated properly to the limit of accuracy
- Optical packet propagated out the correct port at the correct time

S.9.3 Respond to an Environmental Packet in the Optical Switch

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
  - None

S.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

S.9.5 Respond to a Control Message in the Optical Switch

Control Message arrives at the component. Component decodes message properly. Records change in condition or state, if applicable. Change component function if in degraded or damaged state or by change in component condition, if necessary.

- Assumptions
  - Component has completed initialization sequence at least once

S.9.6 Respond to Control Message End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

S.10 MES Switch Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *MEMS Switch Test Cases*

		Inject Ports						Running Totals		
								opt	env	ctrl
Phase	**Case**	**Opt1**	**Opt2**	**Opt3**	**Ctrl**	**Env**	**Notes**	#	#	#
Passive	1	1	0	0	0	0	single	1	0	0
	2	0	1	0	0	0	single	2	0	0
	3	0	0	0	1	0	single	2	0	1
	4	0	0	0	0	1	single	2	1	1
	5	1	1	0	0	0	same time	4	1	1
	6	1	0	0	1	0	same time	5	1	2
	7	1	1	0	0	0	differ time	7	1	2
	8	1	0	0	1	0	differ time	8	1	3
	9	1	1	1	1	1	same time	11	2	4
	10	1	0	1	2	1	differ time	13	3	6
	11	0	1	0	0	1	same time	14	4	6
	12	0	1	0	0	1	differ time	15	5	6
	13	0	0	0	1	1	same time	15	6	7
	14	0	0	0	1	1	differ time	15	7	8
	15	1	0	0	0	1	same time	16	8	8
	16	1	0	0	0	1	differ time	17	9	8
	20	2	0	0	0	0	same time	19	9	8
	21	0	2	0	0	0	same time	21	9	8
	22	2	1	0	0	0	same time	24	9	8
	23	2	0	0	1	0	same time	26	9	9
	24	2	0	0	0	1	same time	28	10	9
	25	2	0	0	1	0	differ time	30	10	10
	26	2	1	0	1	1	same time	33	11	11
	27	2	1	0	1	1	differ time	36	12	12
	28	0	2	0	0	1	same time	38	13	12
	29	0	2	0	0	1	differ time	40	14	12
	30	0	0	0	1	1	same time	40	15	13
	31	0	0	0	1	1	differ time	40	16	14
	32	2	0	0	0	1	same time	42	17	14
	33	2	0	0	0	1	differ time	44	18	14
	34	0	0	1	0	0	single	45	18	14
	35	1	0	1	0	0	differ time	47	18	14
	36	0	1	1	0	0	differ time	49	18	14
	37	0	0	1	1	0	differ time	50	18	15
	38	0	0	1	0	1	differ time	51	19	15
totals		28	16	7	15	19	51			
Respond	41	2	0	0	0	0	single	53	19	15
	42	1	1	0	0	0	single	55	19	15

	43	1	0	0	1	0	single	56	19	16
	44	1	0	0	0	1	single	57	20	16
	45	2	1	1	0	0	same time	61	20	16
	46	2	0	0	1	0	same time	63	20	17
	47	2	0	0	0	1	differ time	65	21	17
	48	2	0	0	1	0	differ time	67	21	18
	49	2	1	1	1	1	same time	71	22	19
	50	2	0	1	2	1	differ time	74	23	21
	51	1	1	0	0	1	same time	76	24	21
	52	1	1	0	0	1	differ time	78	25	21
	60	3	0	0	0	0	same time	81	25	21
	61	1	2	0	0	0	same time	84	25	21
	62	3	1	0	0	0	same time	88	25	21
	63	3	0	0	1	0	same time	91	25	22
	64	3	0	0	0	1	same time	94	26	22
	65	3	0	0	1	0	differ time	97	26	23
	66	3	1	0	1	1	same time	101	27	24
	67	3	1	0	1	1	differ time	105	28	25
	68	1	2	0	0	1	same time	108	29	25
	69	1	2	0	0	1	differ time	111	30	25
	70	1	0	1	0	0	single	113	30	25
	71	1	1	1	0	0	differ time	116	30	25
	72	1	0	1	1	0	differ time	118	30	26
	73	1	0	1	0	1	differ time	120	31	26
	74	1	0	2	0	0	differ time	123	31	26
totals		48	15	9	11	12	72			
	TC1	1	0	0	1	2	single	124	33	27
	TC2	1	0	0	1	2	single	125	35	28
	TC3	1	0	0	1	2	single	126	37	29
	TC4	1	0	0	1	2	single	127	39	30
	TC5	1	0	0	1	2	single	128	41	31
	TC6	1	0	0	1	2	single	129	43	32
	TC7	1	0	0	0	2	single	130	45	32
	TC8	1	0	0	0	2	single	131	47	32
totals		8	0	0	6	16				

Notes: 6- Set to "on" position, port 1 to port 3

10 - Set to "off" position, then opt1, then set to "on" position, then opt3, then env

23 - INIT control message sent; OPT1 & Ctrl - same time - Passive: downstream received packets = 216

25 - INIT control message sent; OPT1 & Ctrl - same time - Passive: downstream received packets = 216

30 - INIT control message sent - Ctrl & ENV - same time - Passive: downstream received

packets = 216

46- Set to "off" position, port 1 to port 2
50 - Send opt1 for Respond phase, set to "off" position, then opt1, then set to "on" position, then opt3, then env
63 - INIT control message sent - OPT1 & Ctrl - same time - Respond: downstream received packets = 209
67 - INIT control message sent - OPT1, OPT2, Ctrl & ENV - differ time - Respond: downstream received packets = 209

S.11 References

DiConFiberOptics. (2013). MEMS 1x2 switch. Retrieved October 04, 2013, Retrieved from http://www.diconfiberoptics.com/products/scd0044/0044h.pdf

Saleh, B. E. A., & Teich, M. C. (1991). *Fundamentals of photonics* (2nd ed.). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). MEMS fiber-optic switches. Retrieved October 04, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1553

Zangari, G. (2010). Micro-electromechanical systems. In M. Schlensinger, & M. Paunovic (Eds.), *Modern electroplating* (5th ed., pp. 617-635). New York: John Wiley & Sons, Inc. Retrieved from http://www2.bren.ucsb.edu/~dturney/port/papers/Modern%20Electroplating/28.pdf

Appendix T - Wave Division Multiplexer (WDM)

T.1 Device Description:

The WDM is an optical device used to combine (or split) different wavelengths of light into one stream. Light from each port enters through a collimation lens, combined using a dichroic filter, and then focused on a collimation lens and output using single mode fiber. In the opposite direction through the WDM, combined optical signals are separated into different streams and sent out different ports (OZOptics, 2013). Fiber-only devices exist that split the beam by wavelength. Signals above a threshold direct to one port, signals below go to the other. A WDM can be made from housing with collimating lenses and some form of a dichroic mirror, a mirror made from thin layers of different transparent optical materials. The different materials constructively interfere to reflect one wavelength and transmit the other (RPPhotonics, 2013). See Figure 1 for an example of a three port fiber-based WDM.

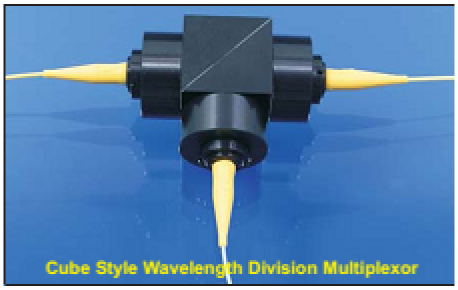


Figure 192. View of a three port WDM (OZOptics, 2013).

Although there are WDMs that can combine many different wavelengths, this document will cover a two-wavelength device that utilizes single-mode input and output fiber, per the discussion with the SME.

The WDM is a bidirectional optical component with three optical ports. In one direction, optical signals arrive at two of the input ports, slightly attenuated and combined for output on the third. In the opposite direction, a single optical input is split into two steams, each stream slightly attenuated, and output to individual ports depending upon wavelength.

Although designed to pass wavelengths within a specified band around a central wavelength (pass width) for each port, there is some undesired throughput to ports 1 and 2 when input on port 3. These are highly attenuated signals, with the attenuation increasing as the undesired frequency moves outside of the pass width range, much like a bandpass filter. Fiber-only devices do not have this undesired throughput.

In the model, this is accounted for by calculating the isolation attenuation for any wavelength outside of the bandwidth, increasing the attenuation until the maximum isolation value is reached for wavelengths well outside the pass width. A check is made on the power of each output optical packet and any that do not have specified minimum amplitude will not be emitted.

Another consideration is the polarization effect of the dichroic mirror material on the optical inputs. If the WDM has single-mode fiber and includes a dichroic mirror, the wavelength that reflects on the mirror will have a π/2 polarization shift along with any shift induced by the single-mode fiber. There are devices that use polarization-maintaining fiber, but as noted earlier, this research will consider only single-mode fiber devices (per discussions with the SME).

The internal material is sensitive to the power of the optical signals that are propagated through the component. If the optical power of a pulse exceeds a defined threshold, the WDM may become permanently damaged which changes its propagation characteristics. Similarly, the WDM is sensitive to the temperature in the environment in which it operates. If the temperature exceeds defined thresholds, the WDM may become temporarily degraded or permanently damaged which changes its propagation characteristics. If temporarily degraded, the device may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with the modeling the WDM is to collect and understand the physical, behavioral, and performance characteristics of the component. In this case, this information was obtained from Subject Matter Expert (SME) with expertise in optical physics. The SME developed a detailed mathematical model in the *Wolfram* Mathematica software program that modeled the WDM. The SME developed a series of use cases that exercised the functionality of the device over a wide variety of conditions and verified the model and validated the input and output behavior of the device within a single Mathematica model (worksheet). The Mathematica worksheet served as the primary means by which the SME communicated the behavior of the WDM to the researcher. Additional information came from product data sheets from commercial vendors and standard texts from the optical field.

The next step of the modeling effort was to develop a conceptual model of the WDM using the DEVS formalism. The bulk of the document following this section is dedicated to the detailed development of the DEVS model of the WDM. Once developed, the model will be simulated using the MS4ME simulator using the same uses cases defined in the Mathematica worksheet. The SME will then review the MS4ME simulation output to verify that the DEVS formal model matches the behavior of the Mathematica model and hence the real component.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

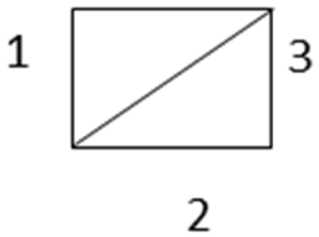


Figure 193. Symbol for the 3-port WDM in the QKD system architecture.

T.2 WDM Conceptual Model

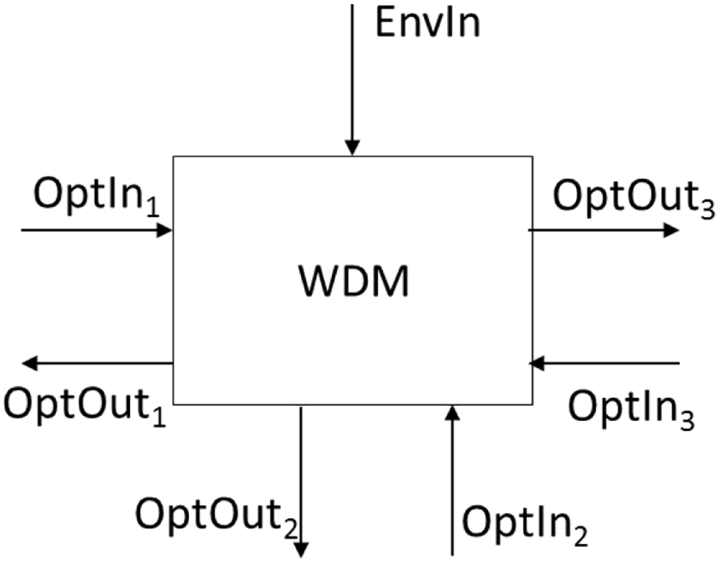


Figure 194. WDM conceptual model.

The conceptual model for a WDM consists of three optical input ports {OptIn$_1$, OptIn$_2$, OptIn$_3$}, three optical output ports {OptOut$_1$, OptOut$_2$, OptOut$_3$}, and one environmental input port {EvnIn}. The environmental port allows external sources to communicate changes in the operational environment to the WDM. In comparison to the WDM symbol used in the QKD simulation architecture shown in Figure 2, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism.

When an optical signal is sent to the input of the WDM, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 3 will instantaneously generate a reflected emitting out of OptOut_1.

The WDM calculates changes to any packet coming through an optical port after a time equaling the propagation delay of the module. The packet is calculated at full power minus some small amount to account for attenuation through the device if passing through to the correct port or heavily attenuated if passing to an incorrect port (for example, passing from port one to port two). The model handles this undesired throughput by splitting each incoming optical packet into a 'strong' packet (one that is output through the correct port) and a 'weak' packet (one that is output through the incorrect port) and injecting these packets into the queue. Each of these entries are a (port, value) pair, just as any other entry into the queue, with the [port] entry equal to the output port and the [value] equal to the adjusted values of the incoming packet. Additionally, packets output on port two rotate $\pi/2$ (i.e. $\alpha = \alpha + \pi/2$) due to the effects of the dichroic mirror inside the device and is applied by the by the 'strong' and 'weak' functions.

The WDM must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters the module. In the case of optical overpowering, once overpowered the device will permanently change attenuation. External environmental messages sent to the device convey the temperature of the operational environmental so the WDM can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the propagation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

T.3 Mathematical Model

For a detailed mathematical description of the WDM, refer to Section 18.8 which contains the Mathematica worksheet provided by the optical physics SME.

T.4 English-Language Rules

In this section, English language rules are developed to express the desired behavior of the WDM.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverPower is a flag which indicates if the device is permanently damaged due to receiving optical signals whose optical power exceed a defined power threshold. Initially, this flag is cleared.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When an optical signal arrives:

- Determine the input port number.
- Immediately calculate the reflected power of the signal and send its output with the same port number.
- Calculate the optical power of the signal. If the optical power exceeds a defined damage threshold, set the OverPower flag.
- Split the incoming optical packet into two entries in the queue.

- Update the values for one queue entry as a ‘strong’ optical signal based on the characteristics of the WDM, the original values of the input optical signal and the current environment and set the correct output port.
- Update the values for the other queue entry as a ‘weak’ optical signal based on the characteristics of the WDM, the original values of the input optical signal and the current environment and set the correct output port.
- After the propagation time has elapsed, send the output signal out of the optical output port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

T.5 Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the WDM in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs. Note there is a self-loop transition from *reflect* to *reflect* if multiple optical packets arrive at the WDM at the same time.

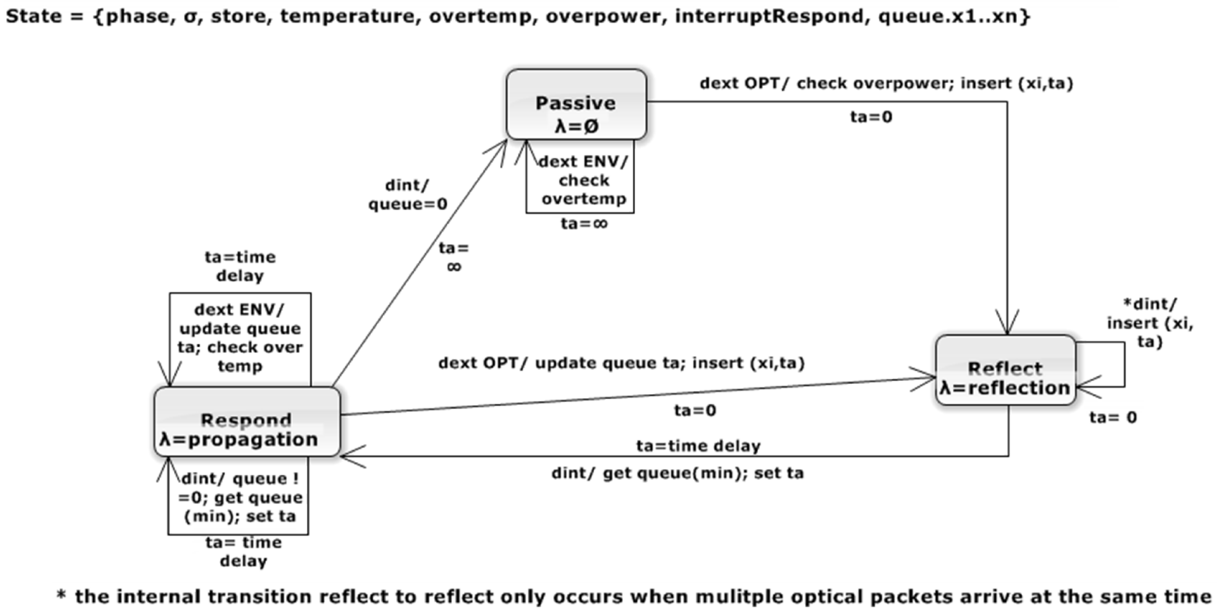

State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
dext OPT/ update queue ta; insert (xi,ta)
ta=0
Reflect
λ=reflection
*dint/ insert (xi, ta)
ta= 0
Respond
λ=propagation
ta=time delay
dint/ get queue(min); set ta
dint/ queue ! =0; get queue (min); set ta
ta= time delay
* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 195. WDM phase transition diagram.

T.6 Event-Trace Diagram

This section shows various examples of packets entering the WDM. The tables list the states the WDM proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the WDM, the over temperature flag variable and the over power flag variable. The next column shows the contents of the queue at that state, the contents of the store state variable and any notes.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Queue: contents of the queue for that state

- Notes: any notes for that state

T.6.1 CASE I: Initial Passive with Single Optical Packet Arriving at Time 0

Table 79. *Case I state list.*

											Notes:
		entry/			**store**		**over**	**over**	**interrupt**	**queue**	**assume**
time	**state**	**exit**	**phase**	**sigma**	**(*xi*)**	**temp**	**temp**	**power**	**respond**	**(*xi*, *tp*)**	**tp=5**
	1-packet	no env	no ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	
0	s2	entry	respond	5	x1	c	n	n	n	null	
5	s2	exit	respond	inf	x1	c	n	n	n	null	
5	s3	entry	passive	inf	x1	c	n	n	n	null	

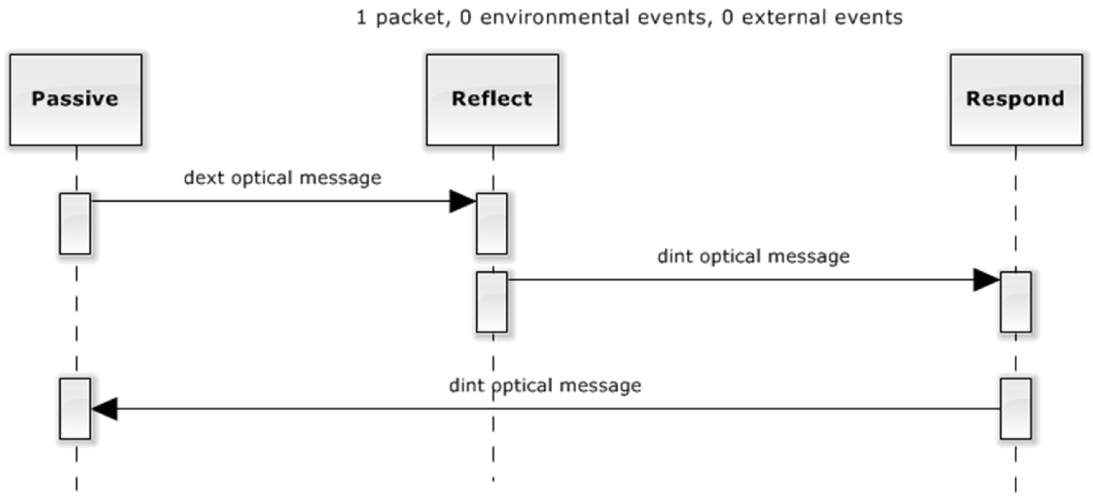


Figure 196. Case I sequence diagram.

T.6.2 CASE II: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2

Table 80. *Case II state list.*

											Notes:
		entry/			**store**		**over**	**over**	**Interrupt**	**queue**	**assume**
time	**state**	**exit**	**phase**	**sigma**	**(*xi*)**	**temp**	**temp**	**power**	**respond**	**(*xi*, *tp*)**	**tp=5**
	1-packet	0 env	1 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
5	s4	exit	respond	0	x2	c	n	n	n	null	
5	s5	entry	respond	2	x2	c	n	n	n	null	
7	s5	exit	respond	inf	x2	c	n	n	n	null	
7	s6	entry	passive	inf	x2	c	n	n	n	null	

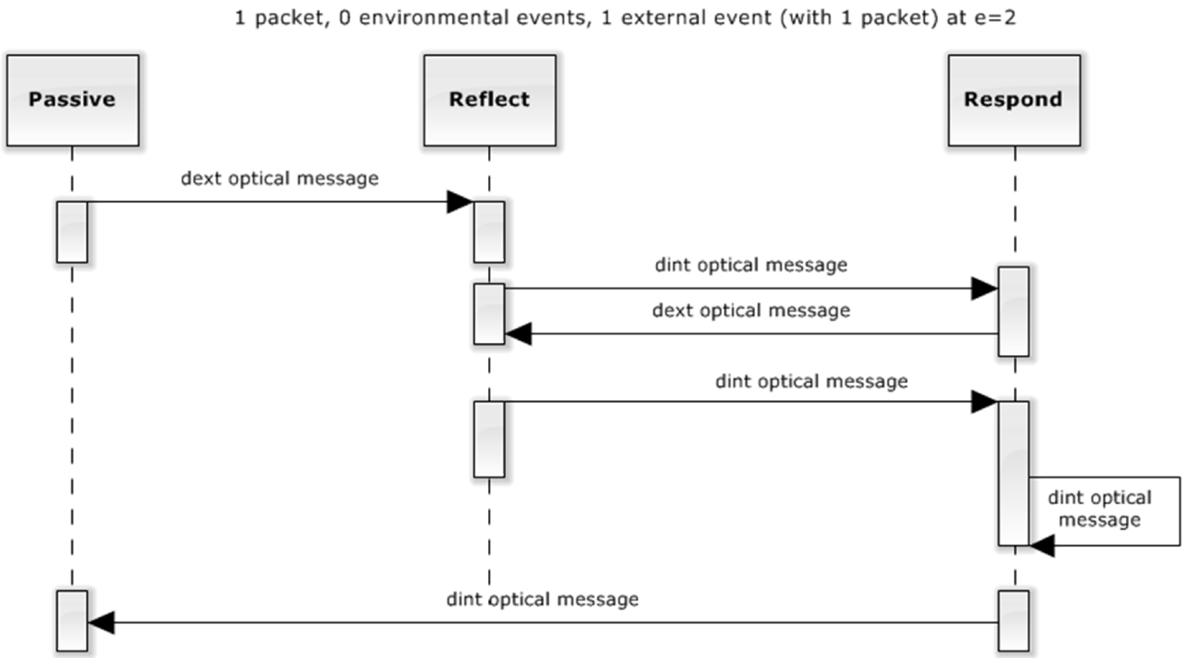


Figure 197. Case II sequence diagram.

T.6.3 CASE III: Initial Passive with Single Optical Packets Arriving at Time 0 and Time 2 and Multiple Optical Packets Arriving at Time 3

Table 81. *Case III state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**interrupt respond**	**queue (*xi*, *tp*)**	**Notes: assume tp=5**
	1-packet	0 env	2 opt	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	
2	s2	exit	respond	0	x1	c	n	n	y	(x2,5)	dext at e= 2, 1 optical packet (x2)
2	s3	entry	reflect	0	x1	c	n	n	y	(x2,5)	
2	s3	exit	reflect	3	x1	c	n	n	y	(x2,5)	
2	s4	entry	respond	3	x1	c	n	n	y	(x2,5)	
3	s4	exit	respond	0	x1	c	n	n	y	(x2,4)(x3,5)	dext at e= 1, 2 optical packets (x3,x4)
3	s5	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)	
3	s5	exit	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	entry	reflect	0	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s6	exit	reflect	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
3	s7	entry	respond	2	x1	c	n	n	y	(x2,4)(x3,5)(x4,5)	
5	s7	exit	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
5	s8	entry	respond	2	x2	c	n	n	n	(x3,3)(x4,3)	
7	s8	exit	respond	1	x3	c	n	n	n	(x4,1)	
7	s9	entry	respond	1	x3	c	n	n	n	(x4,1)	
8	s9	exit	respond	0	x4	c	n	n	n	null	
8	s10	entry	respond	0	x4	c	n	n	n	null	
8	s10	exit	respond	inf	x4	c	n	n	n	null	
8	s11	entry	passive	inf	x4	c	n	n	n	null	

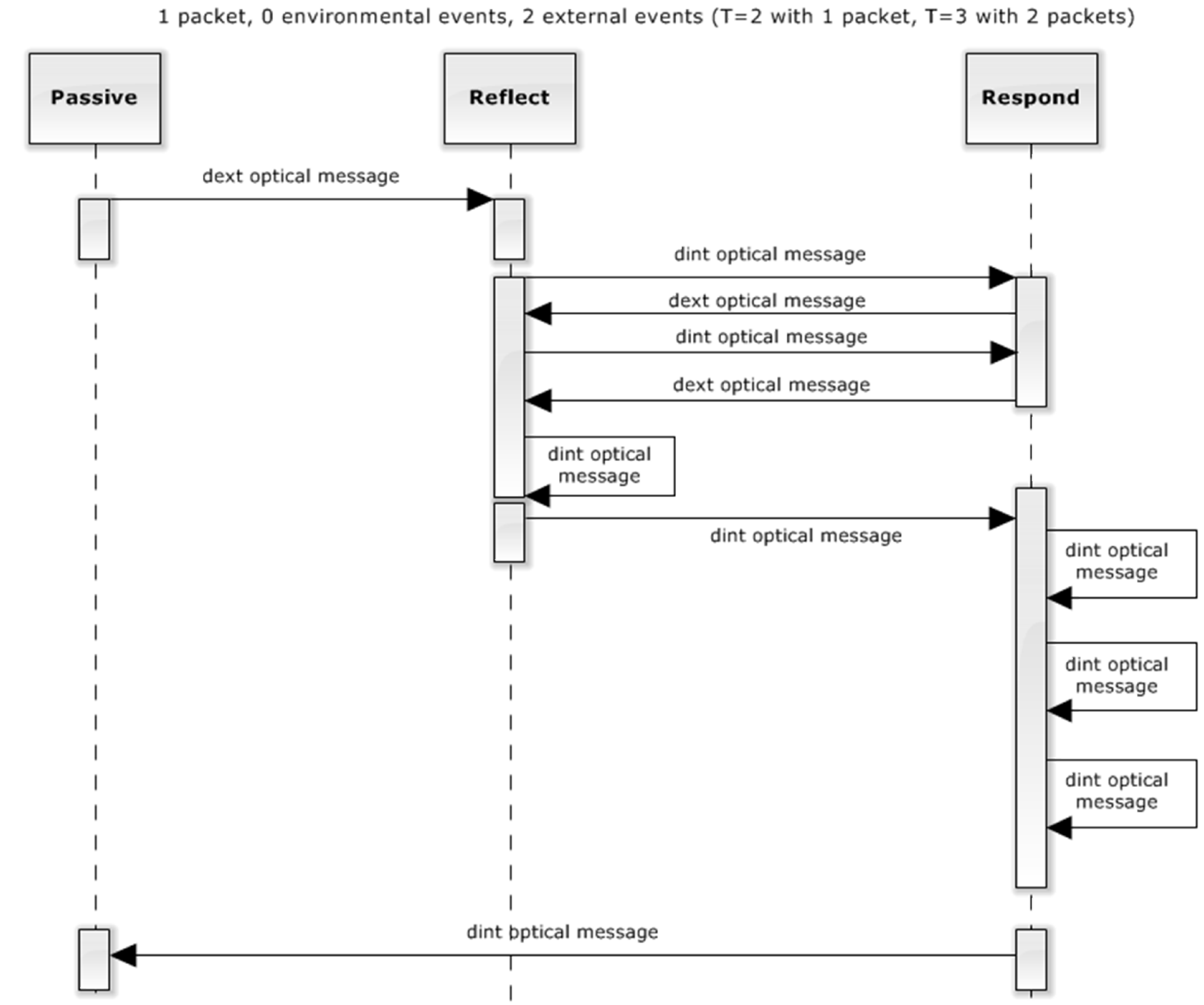


Figure 198. Case III sequence diagram.

T.6.4 CASE IV: Initial Passive with Single Optical Packet Arriving at Time 0 and Single Environmental Packet Arriving at Time 3

Table 82. *Case IV state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	interrupt respond	queue (*xi*, *tp*)	Notes: assume tp=5
	1-packet	1 env	0 ext	0 ctrl							
0	s0	entry	passive	inf	null	c	n	n	n	null	
0	s0	exit	passive	0	null	c	n	n	n	(x1,5)	
0	s1	entry	reflect	0	null	c	n	n	n	(x1,5)	
0	s1	exit	reflect	5	x1	c	n	n	n	null	

0	s2	entry	respond	5	x1	c	n	n	n	null	ENV arrives e=3, overtemp the component
3	s2	exit	respond	2	x1	c	n	n	n	null	update temp
3	s3	entry	respond	2	x1	c	y	n	n	null	
5	s3	exit	respond	inf	x1	c2	y	n		null	
5	s4	entry	passive	inf	x1	c2	y	n		null	

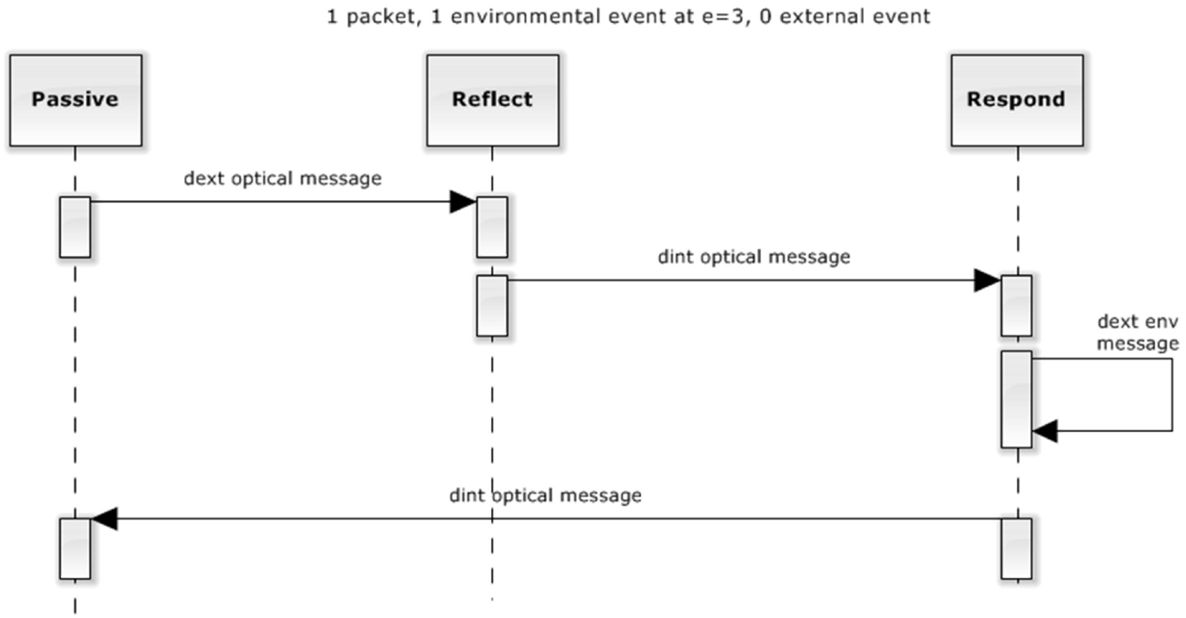


Figure 199. Case IV sequence diagram.

T.7 WDM Parallel DEVS Code

Notes:

- Peak power is calculated as the packet outputs rather than at input due to the small time scale and the short propagation time of the component.
- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “interruptRespond”, queue}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
Peak power = full width, half maximum power calculation of the pulse

For the WDM we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \text{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;
X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS*$_{WDM}$ = (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**
where

t_p = transmission time inside the attenuator
temperature = current temperature of the attenuator
phase = control state that keeps track of the internal phase of the attenuator
phase = {“passive”, “reflect”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event

attenpower = variable the holds the attenuated power of the current optical packet
peak.power = variable the holds the peak power of the current optical packet
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damaged.power = variable that holds the component damaged optical power level parameter
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
need.reflect= variable that stores queue event that needs reflecting
reflect = variable that stores the current reflected optical packet
reflect.port = variable that holds the current reflection output port
reflect.power = variable that holds the current reflection power
time.delay = variable that stores the time delay in the queue for event *i*
output.pulse= variable that stores the output optical packet
output.port = variable that holds the output optical packet port
size= variable that holds the number of events in the queue
new1= variable to hold 1st output values
new2 = variable to hold 2nd output values
queue.current = variable that holds the currently selected queue event
store = variable that holds values of the current optical packet
timeLeftRespond = time left in Respond phase for the current optical packet
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
queue = input container object to store the scheduled inputs
queue_size() = method that returns number of entries in the queue
queue_min() = method that removes the queue entry with the smallest time delay
queue_first() = method that returns the first element of the queue
queue_need_reflected() = method returns the first unreflected queue event
messagebag_first() = method that returns the first element of the message bag
mark_reflected() = method that marks the current queue event as being reflected
update_delay() = method that updates the time delay of entries in the queue by *e*
insert_event_q() = method that inserts the current (x_i, time delay_i) into the queue
remove_event_q() = method that removes the current (x_i, 0) from the queue
remove_event_m() = method that remove the current (x_i, time delay_i) from *messagebag*
calcPeak() = function that calculates full width, half maximum power calculation of the optical pulse
calcAtten() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcStrong() = method that calculates the optical packet high power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcWeak() = method that calculates the optical packet low power output as *f*(*current.v, temperature, overtemp, peakpwr, overpwr*))
calcForward() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcReverse() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)
calcPolar() = method that calculates the optical packet output as: *f*(*store*, *temperature*, *overtemp*, *peakpwr*, *overpwr*)

calcReflected() = method that calculates reflection power of an optical packet
MIN_POWER = global constant that is the minimum acceptable power of an optical packet
q.v = pointer to a value in the queue
q.v_{min} = minimum value in the queue
v.q = value from a queue entry

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"OptIn$_1$", "OptIn$_2$", "OptIn$_3$", "EnvIn"} with
X_M = {("OptIn$_1$", V_{opt}), ("OptIn$_2$", V_{opt}), ("OptIn$_3$", V_{opt}), ("EnvIn", V_{env})} is the set of input ports and values.

OutPorts = {"OptOut$_1$", "OptOut$_2$", "OptOut$_3$"} with
Y_M = {("OptOut$_1$", $Y_{OptOut1}$), ("OptOut$_2$", $Y_{OptOut2}$), ("OptOut$_3$", $Y_{OptOut3}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower interruptRespond, queue*} =
{{"passive", "reflect", "respond"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x {"Y","N"} x *V*}

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, interruptRespond, queue, e, (($p_i$,$v_i$),….
                                                                                   ($p_n$,$v_n$))) =
("reflect", 0, store, temperature, overtemp, overpower, queue.x1..xn)
  if phase = "passive" and p ∈ {"OptIn$_1$", "OptIn$_2$", "OptIn$_3$"}
    for messagebag != null
      current = messagebag_first()
       if current.value.power > damaged.power
         overpower = "Y"
       insert_event_q(current)
       remove_event_m(current)
    queue.current = queue.first(queue)
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond = "N"

("reflect", 0, store, temperature, overtemp, overpower, queue.x1..xn)
if phase = "respond" and p ∈ {"OptIn$_1$", "OptIn$_2$", "OptIn3"}
   update_delay(queue)
    for messagebag != null
     current = messagebag_first()
     if current.value.power > damaged.power
```

```
        overpower =  “Y”
      insert_event_q(current)
       remove_event_m(current)
    queue.current = queue_need_reflected()
    reflect = (queue.current.p), calcReflected(queue.current.v))
    mark_reflected(queue.current)
    interruptRespond= “Y”
   timeLeftRespond = timeLeftRespond - e

(“passive”, ∞, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
   if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

(“respond”, time_delay, store, temperature, overtemp, overpower, interruptRespond,
                                                                    queue.x1..xn)
   if phase = “respond” and p = “EnvIn”
     update_delay(queue)
     timeLeftRespond = time.delay- e
     temperature = messagebag.temperature
     if temperature > damage.temp
       overtemp = “Y”
     time.delay = timeLeftRespond

(phase, σ – e, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue) =
(“reflect”, 0, temperature, overtemp, overpower, interruptRespond, queue.x1..xn))
  if phase = “reflect” and need.reflect != null
    need.reflect = queue_need_reflected()
    current = need.reflect
   reflect = (current.p), calcReflected(current.v))
   mark_reflected(current)


(“respond”, time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
  if phase = “reflect” and need.reflect = null
    need.reflect = queue_need_reflected()
    if interruptRespond = “N”
      current = queue_min()
      time.delay = current.time.delay
    if current.p  = "OptIn$_1$"                  /* input port 1 – strong 3 weak 2 */
     new1 = ("OptOut3",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
     new2 = ("OptOut2",calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
    else
      if current.p  = "OptIn$_2$"                /* input port 2 – strong 3 weak 1 */
        new1 = ("OptOut3",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
        new2 = ("OptOut1",calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
      else                                       /* input port 3 – strong 1 strong 2*/
        new1 = ("OptOut1",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
        new2 = ("OptOut2",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
    timeLeftRespond = propagation delay
   else
     time.delay = timeLeftRespond

 ("respond", time.delay, store, temperature, overtemp, overpower, interruptRespond,
queue.x1..xn)
    if phase = "respond" and size > 0
      update_delay(queue)
      size= queue_size()
      current = queue_min()
      time.delay = current.time.delay
      if current.p  = "OptIn$_1$"                /* input port 1 – strong 3 weak 2 */
       new1 = ("OptOut3",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
       new2 = ("OptOut2",calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
      else
        if current.p  = "OptIn$_2$"              /* input port 2 – strong 3 weak 1 */
          new1 = ("OptOut3",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
          new2 = ("OptOut1",calcWeak(current.v, temperature, overtemp, peakpwr, overpwr))
        else                                     /* input port 3 – strong 1 strong 2*/
          new1 = ("OptOut1",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
          new2 = ("OptOut2",calcStrong(current.v, temperature, overtemp, peakpwr, overpwr))
       interruptRespond= "N"

 ("passive", $\infty$, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn)
    if phase = "respond" and size  = 0
      size= queue_size()
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:

```
λ(phase, σ, store, temperature, overtemp, overpower) =
   (reflect.p, reflect.v)
      if phase = "reflect"
```

$(new1.p, new1.v)$
if phase = “respond”

$(new2.p, new2.v)$
if phase = “respond”

∅ (null output)
otherwise;

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower) = \sigma$;

T.8 Mathematical Model

Pulse propagation considerations for the Dichroic Mirror (Wavelength Division Multiplexer) within the QKD OMNet++ simulation environment

The dirchroic mirror (wavelength division multiplexer) is a passive device which combines, in a low-loss format, two input wavelengths into a single fiber. In reverse, the device functions to separate two co-propagating wavelengths onto separate fibers. They are frequently used for add/drop filters, erbium doped fiber amplifier (EDFA) pumping, etc., in the telecommunications industry. The devices have many formats, from free-space dichroic mirrors (using collimating elements to launch from and focus onto the fiber ends), to in-line fiber devices. Though integrated devices exist which can combine many wavelengths, we will here consider two wavelengths, namely 1310 and 1550 nm. In both applications (Alice and Bob) the WDM requires single-mode input and output fiber - this is reflected in the model below.

The primary operational characteristics are as follows:

- light input to **port 1** (at wavelength 1) will exit **port 3**
- light input to **port 2** (at wavelength 2) will exit **port 3**
- light input to **port 3** (at wavelength 1) will exit **port 1**
- light input to **port 3** (at wavelength 2) will exit **port 2**

The only significant modification to the optical message will be the amplitude (power).

Pulse Characteristics (e.g.)

These parameters are used in the jones representation of the standard coherent pulse optical message packet.

$$\mathrm{E(t)} = \begin{pmatrix} E_x \\ E_y \end{pmatrix} = g(t)\,\mathrm{Eo}\,e^{i\,\omega_o t}\,e^{i\,\theta}\begin{pmatrix} \cos\alpha \\ (\sin\alpha)\,e^{i\,\phi} \end{pmatrix}$$

Pertinent Pulse Characteristics for the Optical Switch Module

```
Ein1 := Eo (* electric field input into port 1 *)
ωo1 := ω1(* optical frequency input to port 1, units of rad/s *)

Ein2 := Eo (* electric field input into port 2 *)
ωo2 := ω3 (* optical frequency input to port 2, units of rad/s *)

Ein3 := Eo (* electric field input into port 3 *)
ωo3 := ω3 (* optical frequency input to port 3, units of rad/s *)
```

The following parameter values are examples of a typical optical fiber-based WDM and are taken from COTS "Miniature Inline" 60dB return-loss devices offered by Oz Optics (http://www.ozoptics.com/ALLNEW_PDF/DTS0089.pdf).

Optical Specifications

```
InsertionLoss := 0.7 (* maximum insertion loss, units of -dB *)
Isolation := 20
(* maximum power throughput to the incorrect port (from port 3), units of -dB *)
RetLoss := 60 (* maximum return loss,
signal reflected by an input beam, units of -dB *)
TempH := 60 (* max operational temperature, units of °C *)
TempL := -20 (* min operational temperature, units of °C *)
MaxPwr := 200 (* maximum operational optical power, units of mW *)
WavelengthPass1 := 1550 * 10^-9 (* wavelength passed from port 1 → port 3,
port 3 → port 1, units of m *)
WavelengthPass2 := 1310 * 10^-9 (* wavelength passed from port 2 → port 3,
port 3 → port 2, units of m *)
WavelengthPassWidth := 10 * 10^-9 (* width of pass widow
   (from specification) without significant loss, units of +/- m *)
```

Attenuation Calculations for the 2 input WDM

First, we must calculate the input wavelengths

```
c := 2.99792458 * 10^8 (* speed of light, units of m/s *)
```

$\lambda 1 := \frac{2\pi c}{\omega o1}$ `(* wavelength input to port 1, units of m *)`

$\lambda 2 := \frac{2\pi c}{\omega o2}$ `(* wavelength input to port 2, units of m *)`

$\lambda 3 := \frac{2\pi c}{\omega o3}$ `(* wavelength input to port 3, units of m *)`

Input Port 1:

Pseudocode:

if λ1 ≤ WavelengthPass1+WavelengthPassWidth && λ1 ≥ WavelengthPass1-WavelengthPassWidth

$\text{Eout3[Ein1\_, InsertionLoss\_] := Ein1} * \sqrt{10^{-\text{InsertionLoss}/10}}$

```
(* case: optical input on port 1, λ within pass window *)
ωout3[ωo1_] := ωo1 (* case: optical input on port 1, λ1 within pass window *)
```

else

```
Eout3 := 0 (* case: optical input on port 1, λ1 out of pass window, message killed *)
(* although this isn't fully phycial, until we have an appropriate model for out-
 of-band wavelengths we will kill the message *)
```

Input Port 2:

Pseudocode:

if λ2 ≤ WavelengthPass2+WavelengthPassWidth && λ2 ≥ WavelengthPass2-WavelengthPassWidth

$\text{Eout3[Ein2\_, InsertionLoss\_] := Ein2} * \sqrt{10^{-\text{InsertionLoss}/10}}$

```
(* case: optical input on port 2, λ2 within pass window *)
ωout3[ωo2_] := ωo2 (* case: optical input on port 2, λ within pass window *)
```

else

```
Eout3 := 0 (* case: optical input on port 2, λ2 out of pass window, message killed *)
(* although this isn't fully phycial, until we have an appropriate model for out-
 of-band wavelengths we will kill the message *)
```

Input Port 3:

Pseudocode:

if λ3≤ **WavelengthPass1+WavelengthPassWidth** && λ3 ≥ **WavelengthPass1-WavelengthPassWidth**

```
Eout1[Ein3_, InsertionLoss_] := Ein3 * $\sqrt{10^{-InsertionLoss/10}}$
(* case: optical input on port 3, λ3 within port 1 pass window *)
ωout1[ωo3_] := ωo3 (* case: optical input on port 3, λ3 within port 1 pass window *)

(* we may wish to create a flag to allow undesired throughput *)
Eout2[Ein3_, InsertionLoss_, Isolation_] := Ein3 * $\sqrt{10^{-(InsertionLoss+Isolation)/10}}$
(* case: optical input on port 3,
λ3 within port 1 pass window, incorrect port 2 pass *)
ωout2[ωo3_] := ωo3 (* case: optical input on port 3,
 λ3 within port 1 pass window, incorrect port 2 pass *)
```

else if λ3≤ **WavelengthPass2+WavelengthPassWidth** && λ3 ≥ **WavelengthPass2-WavelengthPassWidth**

```
Eout2[Ein3_, InsertionLoss_] := Ein3 * $\sqrt{10^{-InsertionLoss/10}}$
(* case: optical input on port 3, λ3 within port 2 pass window *)
ωout2[ωo3_] := ωo3 (* case: optical input on port 3, λ3 within port 2 pass window *)

(* we may wish to create a flag to allow undesired throughput *)
Eout1[Ein3_, InsetionLoss_, Isolation_] := Ein3 * $\sqrt{10^{-(InsertionLoss+Isolation)/10}}$
(* case: optical input on port 3,
λ3 within port 2 pass window, incorrect port 1 pass *)
ωout1[ωo3_] := ωo3 (* case: optical input on port 3,
 λ3 within port 2 pass window, incorrect port 1 pass *)
```

else

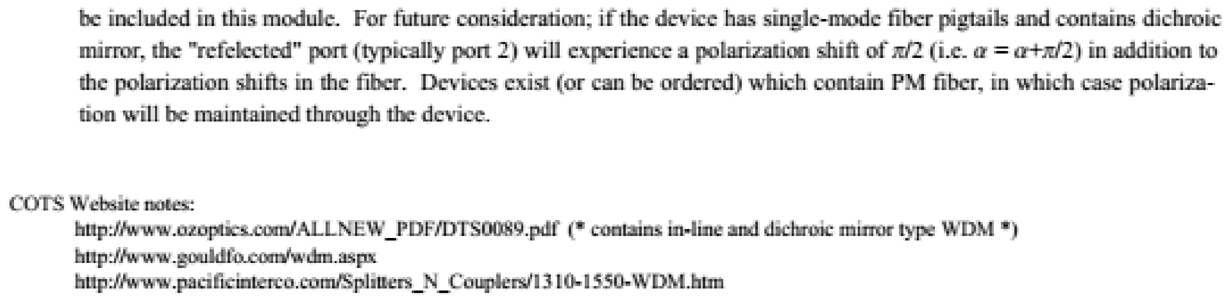
```
Eout1 := 0 (* case: optical input on port 3,
λ out of port 1 and port 2 pass window2, message killed *)
Eout2 := 0 (* case: optical input on port 3,
λ out of port 1 and port 2 pass window2, message killed *)
(* although this isn't fully phycial, until we have an appropriate model for out-
 of-band wavelengths we will kill the message *)
```

If we wish to flag the 2 Input WDM to include **undesired return (reflected)** messages, the following operations would hold true,

```
Eout1[Ein1_, RetLoss_] := Ein1 * $\sqrt{10^{-RetLoss/10}}$

Eout2[Ein2_, RetLoss_] := Ein2 * $\sqrt{10^{-RetLoss/10}}$

Eout3[Ein3_, RetLoss_] := Ein3 * $\sqrt{10^{-RetLoss/10}}$
```

Polarizaion Considerations for 2 Input WDM

The polariztion change for an opitcal message transiting the WDM can be related to that of a similar length of fiber, in this case SMF-28. This will be incorporated into the system polarization state randomization and drift, and so will not



be included in this module. For future consideration; if the device has single-mode fiber pigtails and contains dichroic mirror, the "refelected" port (typically port 2) will experience a polarization shift of $\pi/2$ (i.e. $\alpha = \alpha+\pi/2$) in addition to the polarization shifts in the fiber. Devices exist (or can be ordered) which contain PM fiber, in which case polarization will be maintained through the device.

COTS Website notes:
http://www.ozoptics.com/ALLNEW_PDF/DTS0089.pdf (* contains in-line and dichroic mirror type WDM *)
http://www.gouldfo.com/wdm.aspx
http://www.pacificinterco.com/Splitters_N_Couplers/1310-1550-WDM.htm

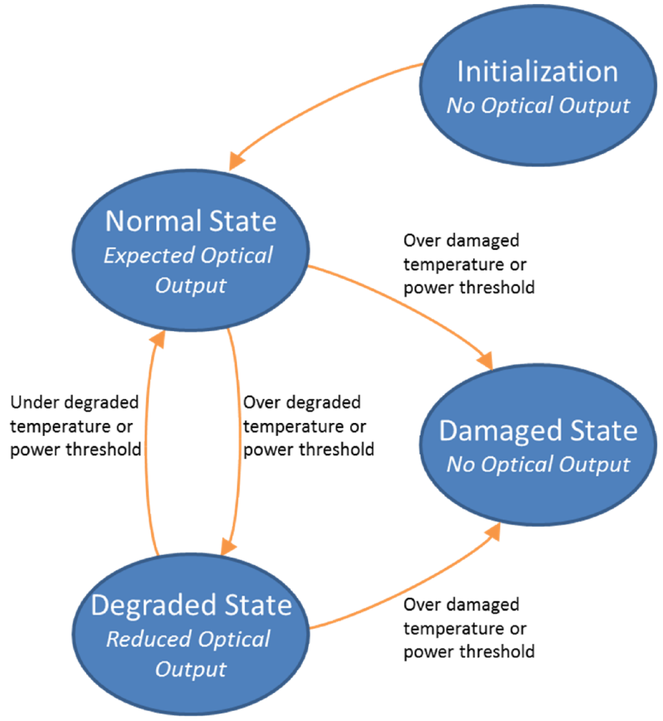
T.9 Component Use Case

T.9.1 Respond to an Optical Packet in the Wave Division Multiplexer (WDM)

Optical packet arrives at the WDM. A portion of optical packet reflects back down incoming optical line. Place the optical packet into the optical queue. Check to see if optical packet overpowers the WDM. Records overpower condition, if applicable. Remove the optical packet from the queue and create transmitted and reflected packets. Calculate the attenuated optical output signal based on the input signal, whether transmitted or reflected, and the current component state. Propagate the attenuated optical output signals out of the component optical ports based on the input port and being transmitted or reflected.

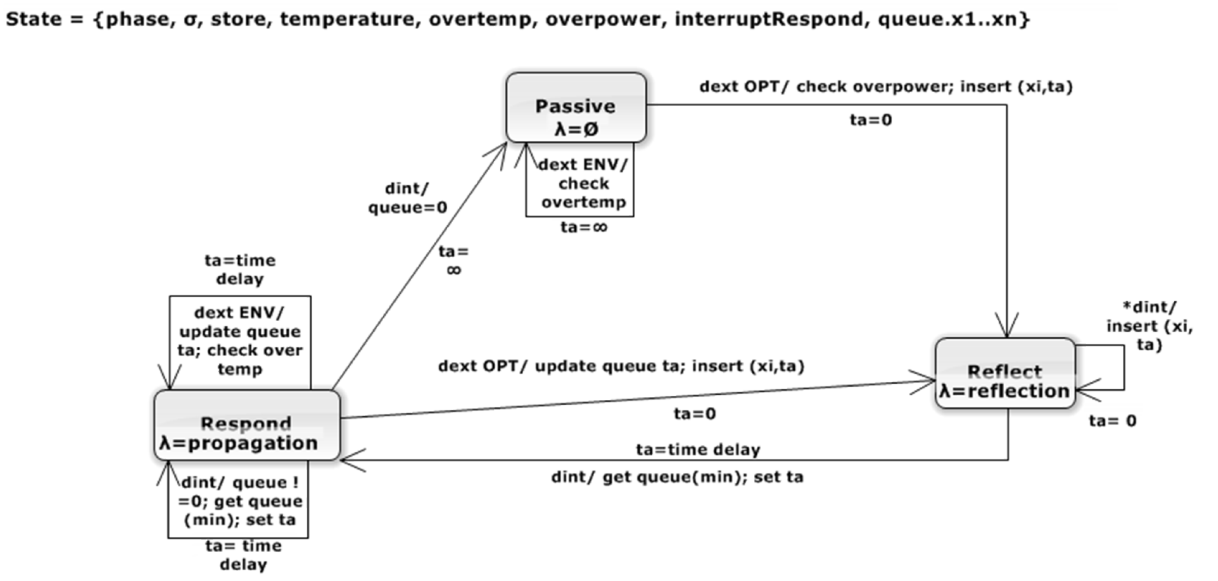
- Identified Alternative Uses Cases
  - React to an environmental message

- Assumptions
  - Component has completed initialization sequence at least once
  - Reflections are not affected by component state
  - Incoming electrical signals are not affected by component state


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 200. Component states.


State = {phase, σ, store, temperature, overtemp, overpower, interruptRespond, queue.x1..xn}
Passive
λ=Ø
dext OPT/ check overpower; insert (xi,ta)
ta=0
dext ENV/ check overtemp
ta=∞
dint/ queue=0
ta= ∞
ta=time delay
dext ENV/ update queue ta; check over temp
*dint/ insert (xi, ta)
dext OPT/ update queue ta; insert (xi,ta)
Reflect
λ=reflection
Respond
λ=propagation
ta=0
ta= 0
ta=time delay
dint/ get queue(min); set ta
dint/ queue != 0; get queue (min); set ta
ta= time delay

* the internal transition reflect to reflect only occurs when mulitple optical packets arrive at the same time

Figure 201. WDM phase transition diagram.

T.9.2 Respond to Optical Packet End Goals

- Optical packet reflected properly
- Optical packet entered and removed from queue in proper sequence

- Overpower condition properly recognized and recorded
- Optical packet attenuated properly to the limit of accuracy
- Optical packet propagated out the correct port at the correct time

T.9.3 Respond to an Environmental Packet in the Wave Division Multiplexer (WDM)

Environmental packet arrives at the component. Check to see if environmental packet temperature sets the component to degraded or damaged state. Check to see if temperature level returns component from degraded state to normal state. Records change in condition, if applicable. Change component function if in degraded or damaged state.

- Assumptions
    - None

T.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

T.10 WDM Test Cases

Each optical component was tested by sending inputs into the component, capturing the output, and evaluating the output line-by-line to check behavior and timing. Each component had each of its input ports (optical, environmental (env), and/or control (ctrl)) tested singly, then in different combinations of ports and input messages. All identified errors were corrected and the component retested until it functioned properly for each test case.

To test an optical port, an optical message is injected into that port when the component is in Passive or Respond phase. This tests component behavior when it is do nothing and awaiting input or the behavior when the component is interrupted during message processing. Control messages work in the same way, but force the component to begin behavior to react to the contents of the messages. Environmental packets force an immediate response to the change

in temperature, possibly changing the properties of the component if it is damaged or degraded by the new temperature.

The following table summarizes these tests by listing the component on the left and the number and type of tests across the top. Each component is in either the Passive or Respond phase when reacting to inputs as noted at the top of each table. Each box shows the number of tests exercising the particular type of port. The first column lists the total number of tests performed on a component; successive columns list the number of those tests that exercise a particular port (optical, ctrl, or env) and the number of single or multi-port tests, with the final column listing the number of math-specific tests. These math tests were created by the optical SME to exercise the early demonstration QKD simulation and added in the MS4ME code for possible future work in comparing the conceptual models to the *qkdX* framework.

Table 5. *WDM Test Cases.*

		Inject Ports					Running Totals	
Phase	**Case**	**Opt1**	**Opt2**	**Opt3**	**Env**	**Notes**	**opt #**	**env #**
Passive	1	1	0	0	0	single	1	0
	2	0	1	0	0	single	2	0
	3	0	0	1	0	single	3	0
	4	0	0	0	1	single	3	1
	5	1	1	1	0	same time	6	1
	6	1	1	1	0	differ time	9	1
	7	1	1	1	1	same time	12	2
	8	1	1	1	1	differ time	15	3
	9	0	1	0	1	same time	16	4
	10	0	1	0	1	differ time	17	5
	11	1	0	0	1	same time	18	6
	12	1	0	0	1	differ time	19	7
	13	0	0	1	1	same time	20	8
	14	0	0	1	1	differ time	21	9
	20	2	0	0	0	same time	23	9
	21	0	2	0	0	same time	25	9
	22	0	0	2	0	same time	27	9
	23	2	2	2	0	same time	33	9
	24	2	2	2	0	differ time	39	9

	25	2	2	2	1	same time	45	10
	26	2	2	2	1	differ time	51	11
	27	0	2	0	1	same time	53	12
	28	0	2	0	1	differ time	55	13
	29	2	0	0	1	same time	57	14
	30	2	0	0	1	differ time	59	15
	31	0	0	2	1	same time	61	16
	32	0	0	2	1	differ time	63	17
totals		21	21	21	17	63		
Respond	41	2	0	0	0	single	65	17
	42	0	2	0	0	single	67	17
	43	0	0	2	0	single	69	17
	44	1	0	0	1	single	70	18
	45	2	1	1	0	same time	74	18
	46	2	1	1	0	differ time	78	18
	47	2	1	1	1	same time	82	19
	48	2	1	1	1	differ time	86	20
	49	0	2	0	1	same time	88	21
	50	0	2	0	1	differ time	90	22
	51	2	0	0	1	same time	92	23
	52	2	0	0	1	differ time	94	24
	53	0	0	2	1	same time	96	25
	54	0	0	2	1	differ time	98	26
	60	3	0	0	0	same time	101	26
	61	0	3	0	0	same time	104	26
	62	0	0	3	0	same time	107	26
	63	3	2	2	0	same time	114	26
	64	3	2	2	0	differ time	121	26
	65	3	2	2	1	same time	128	27
	66	3	2	2	1	differ time	135	28
	67	0	3	0	1	same time	138	29
	68	0	3	0	1	differ time	141	30
	69	3	0	0	1	same time	144	31
	70	3	0	0	1	differ time	147	32
	71	0	0	3	1	same time	150	33
	72	0	0	3	1	differ time	153	34
totals		36	27	27	17	90		
	TC1	1	0	0	2	single	154	36
	TC2	1	0	0	2	single	155	38
	TC3	1	0	0	2	single	156	40
	TC4	1	0	0	2	single	157	42
	TC5	1	0	0	2	single	158	44
	TC6	1	0	0	2	single	159	46

	TC7	1	0	0	2	single	160	48
totals		7	0	0	14	7		

T.11 References

OZOptics. (2013). Wave division multiplexers. Retrieved September 24, 2013, Retrieved from http://www.ozoptics.com/ALLNEW_PDF/DTS0089.pdf

RPPhotonics. (2013). RP phontics encylopedia - dichroic mirrors. Retrieved September 24, 2013, Retrieved from http://www.rp-photonics.com/dichroic_mirrors.html

Appendix U - Classical Pulse Generator (CPG)

U.1 Device Description:

The ideal conceptual model of a QKD system specifies polarization-encoded single photons with the desired bit and basis. In reality, reliable on-demand single photon pulse generators are an unrealized technology. Real-world QKD system implementations instead generate a laser pulse containing millions of photons and strongly attenuate the pulse down to statistical sub-photon (quantum) levels. Within the Alice quantum module, the CPG subsystem generates the laser pulses and shifts them into a known polarization. The CPG subsystem contains the components shown in Fig. 1.

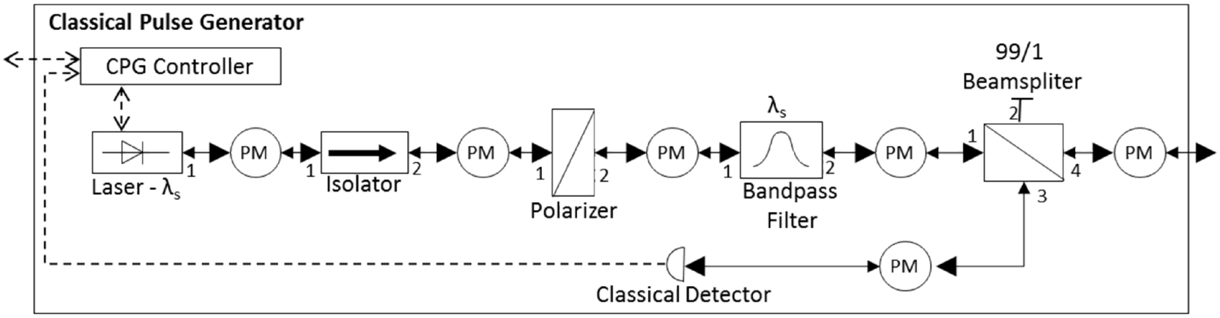


Figure 202. Classical Pulse Generator (CPG) in the QKD system architecture.

The CPG subsystem contains a controller, a laser, an isolator, an optical polarizer, an optical bandpass filter, a beamsplitter, a classical detector, electrical interfaces, and interconnecting polarization-maintaining (PM) optical fiber.

propagation delay.

U.1.1 CPG Controller

The controller is an electrical device containing digital and analog circuits responsible for controlling the laser and monitoring the classical detector. It has a bidirectional electrical interface to the quantum module controller, an electrical output to the laser, and an electrical input from the classical detector. It receives commands from the quantum model controller, sends fire commands to the laser, and monitors the health of the laser.

U.1.2 Laser

The laser is an electro-optical device which contains an optical oscillator and emits coherent light (Saleh & Teich, 1991). It has an electrical input to receive control messages and an optical output to emit generated pulses. Within the simulation, the laser creates optical pulses when it receives a "fire" command from the controller. The laser generates short-duration laser pulses (e.g., 1mW peak intensity with a 500ps duration) containing millions of photons (ThorLabs, 2013b). The output of the laser couples to the input of the isolator via PM fiber.

U.1.3 Isolator

The isolator is an optical device with two bidirectional optical ports that passes light in the forward direction while significantly attenuating light moving in the opposite direction (ThorLabs, 2013d). Optical signals arriving at one port propagate to the other port after a defined propagation delay with the attenuation based on the propagation direction. The isolator assures that virtually no light (e.g., reflections or light from external sources) enters the laser. The output of the isolator is coupled to the input of the polarizer via PM fiber

U.1.4 Polarizer

The polarizer is an optical device with two bidirectional optical ports allowing light of one polarization to pass while highly attenuating light orthogonal to the passed light (ThorLabs, 2013c). Optical signals arriving at one port propagate to the other port after a defined propagation delay and polarized depending on the polarizer orientation with respect to the connected fiber. The output of the polarizer is coupled to the input of the optical bandpass filter via PM fiber.

U.1.5 Bandpass Filter

The bandpass filter is an optical device with two bidirectional optical ports that passes the optical energy in a narrow band around the signal wavelength, λ_S, but strongly attenuates other wavelengths (ThorLabs, 2013a). This ensures that only the appropriate signal wavelength leaves the subsystem while preventing other sources of light from entering the laser. Optical signals arriving at one port propagate to the other port after a defined propagation delay and are attenuated based on the wavelength of the signal. The bandpass filter output couples to port 1 of the beamsplitter.

U.1.6 Beamsplitter

The beamsplitter is an optical device used to split a single beam of light into two components. It can also be used to combine two beams of light into one stream (OZOptics, 2013). Unlike most of the optical devices, it has four bidirectional optical ports. In the splitting configuration, optical signals arriving at one port are split into two beams, propagating to the appropriate output ports after a defined propagation delay. Common splitting ratios are 50:50, 90:10, and 99:1, but devices exist in almost any ratio. Beams can also be split according to optical wavelength or polarization. The beamsplitter passes 99% of the pulse through to port 4, leaving the CPG and connecting to the next quantum module subsystem as shown in Fig. 9. Meanwhile, port 3 passes 1% of the pulse on to the classical detector via PM fiber.

U.1.7 Classical Detector

The classical detector is an opto-electrical device containing an optical photodiode and support electronics to generate an electrical signal proportional to the power contained in the optical pulse (ThorLabs, 2013e). This signal connects to the controller which stores this information and checks to see if it falls below a predefined threshold. If so, the controller notifies the quantum module controller of an error condition.

U.1.8 Polarization-Maintaining Optical Fiber

PM fiber is a specialty fiber that intentionally uses the strong birefringence in two modes. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass, and has two bidirectional optical ports. Light travels down one of the modes faster than down the other (fast and slow axes). If the input light is polarized and oriented along either mode, it maintains its polarization state even if the fiber is stressed (OZOptics, 2014). Typically, PM fiber is used in components that cannot have drift in the polarization state (such as fiber interferometers and some fiber lasers) (RPPhotonics, 2013). Optical signals arriving at one port propagate to the other port after a defined propagation delay.

U.2 CPG and Controller Behavior

The controller and individual components are sensitive to the temperature in the environment in which they operate. If the temperature exceeds defined thresholds, the components may become temporarily degraded or permanently damaged which changes their characteristics. If temporarily degraded, the devices may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with modeling the controller and CPG is to collect and understand the physical, behavioral, and performance characteristics of the atomic components. In this case, the individual components were constructed earlier and the controller was built as a message handler. The logic for the controller was based on the types of messages necessary for control of components inside the module.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

U.3 CPG Compound Conceptual Model

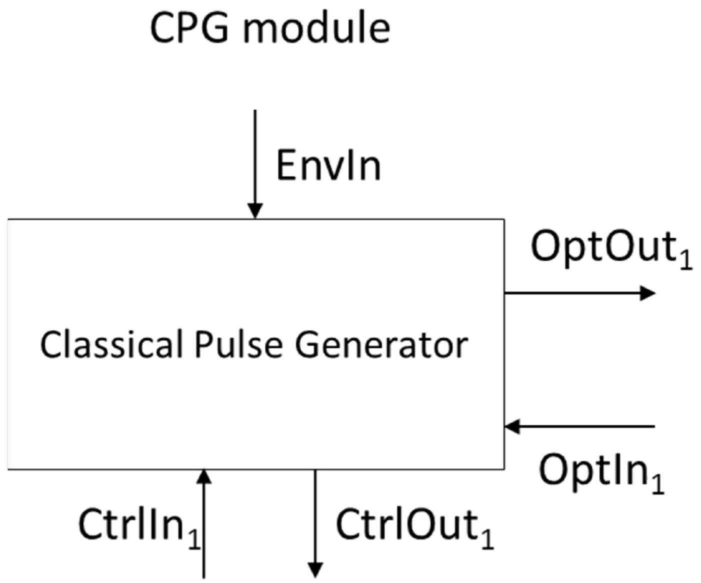


Figure 203. Classical Pulse Generator (CPG) compound module conceptual model.

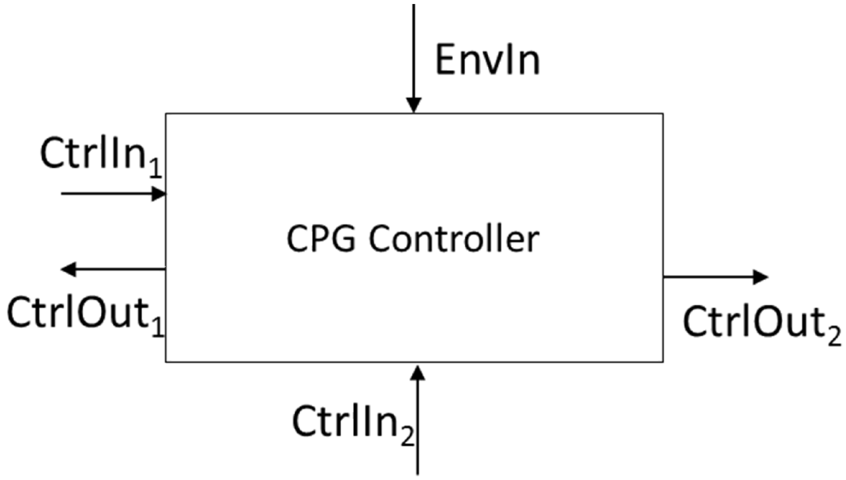


Figure 204. Classical Pulse Generator (CPG) controller conceptual model

Table 83. *List of CPG Controller messages*

Input Messages	**From**	**Response**
CPG_ENV	Quantum controller	Set the internal CPG controller temperature
CPG_RESET	Quantum controller	Resets the CPG controller and clears the state variables
CPG_STATUS_REQUEST	Quantum controller	Sends the CPG controller status and stored magnitude value
CPG_FIRE_LASER	Quantum controller	Issues a single Fire Laser command to the laser
CD_DETECTION	Classical Detector	Store the magnitude from the message
Output Messages	**To**	**Content**
CPG_ACK	Quantum Controller	Response to a Reset message
CPG_STATUS	Quantum Controller	Response to a Status Request message
CPG_LASER_FIRE	Laser	Command to fire the laser one time

The conceptual model for a CPG consists of one optical input port {OptIn$_1$}, one optical output port {OptOut$_1$}, one environmental input port {EvnIn}, one control input port {CtrlIn} and one control output port {CtrlOut}. The environmental port allows external sources to communicate changes in the operational environment to the CPG. The electrical controller ports allow for control inputs to the controller and responses from the CPG to the higher system functions.

In comparison to the CPG layout used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism. The electrical control port is also decomposed in the model into an input port and an output port.

When an optical signal is sent to the input of the CPG, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The CPG components must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters each of the components the module. In the case of optical overpowering, once overpowered a component will permanently change attenuation. External environmental messages sent to the CPG are directed to individual components convey the temperature of the operational environmental so the module can determine if it is degraded (a temporary condition) or damaged (a permanent condition). In either case, a function determines how the attenuation changes as a function of the device state and current temperature.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation.

This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

U.4 English-Language Rules for the Controller

In this section, English language rules are developed to express the desired behavior of the controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.

- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When a control signal arrives:

- Determine the arrival port of the signal.
- Evaluate the content of the message
- Generate a response message to the incoming signal (if necessary).
- Generate a forwarded message to the appropriate device (if necessary).
- Output the response or forwarded message out the appropriate port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

U.5 DEVS Phase Transition Diagram (Controller)

The controller phase transition diagram in Fig. 4 shows the phases of the CPG controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs.

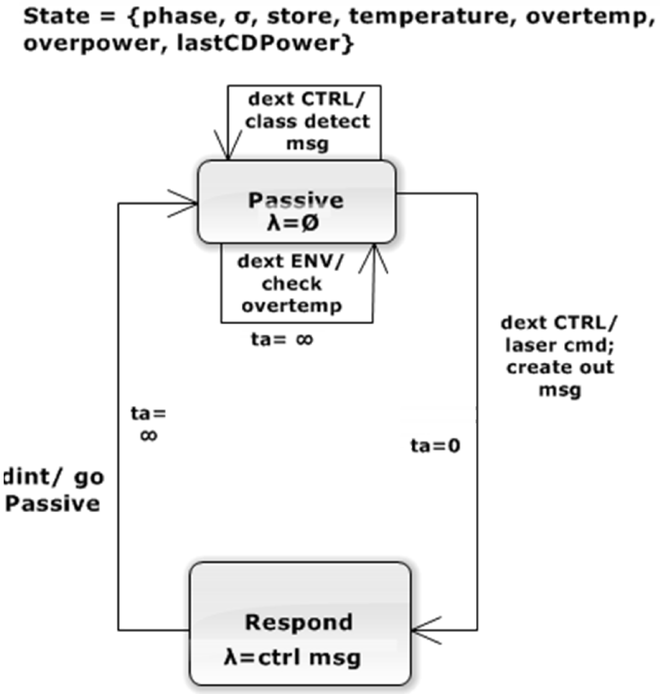

State = {phase, σ, store, temperature, overtemp, overpower, lastCDPower}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ laser cmd; create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 205. CPG Controller DEVS phase transition diagram

U.6 CPG Controller Event-Trace Diagram

This section shows various examples of packets entering the CPG controller. The tables list the states the component proceeds through as the packets are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the component, the over temperature flag variable and the over power flag variable. The queue column shows the contents of the queue at that state, the contents of the store state variable and any notes. Note in contrast to most other components, the controller is very simple and only responds to incoming messages; it does not generate any messages on its own. There are two types of inputs: control messages and environmental messages.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- LastCDPower: shows the power of the last classical detection message
- Notes: any notes for that state

U.6.1 CASE I: Initial Passive with Single Control Packet (Fire) Arriving at Time 0

Table 84. *Case I state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	lastCD power	Notes: assume tp=0
	1-packet	no env	no ext	1 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	s1	entry	respond	0	null	c	n	n	null	
0	s1	exit	respond	inf	null	c	n	n	null	
0	s2	entry	passive	inf	null	c	n	n	null	

U.6.2 CASE II: Initial Passive with Single Environmental Packet Arriving at Time 0

Table 85. *Case II state list*.

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	lastCD power	Notes: assume tp=5
	1-packet	1 env	no ext	0 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	S1	entry	passive	inf	null	c	n	n	null	

U.7 Classical Pulse Generator (CPG) Controller Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “lastCDPower”}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
“lastCDPower” = variable to store the power value of the last classical detector message

For the CPG controller we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid \text{p} \in \mathit{InPorts}, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in \mathit{OutPorts}, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

DEVS$_{\boldsymbol{CPGcontroller}}$ **=** $\boldsymbol{(X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$

where

t_p = transmission time inside the component
temperature = current temperature of the component
phase = control state that keeps track of the internal phase of the component
phase = {“passive”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
lastCDPower = variable that holds the power value of the last classical detector message

interruptRespond = flag variable set when Respond phase is interrupted by an external event
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
ctrlOutput = variable that stores the output control message response
output.port = variable that holds the output optical packet port
store = variable that holds values of the current input values
timeLeftRespond = time left in Respond phase for the current event
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
messagebag_first() = method that returns the first element of the message bag
remove_event_m() = method that remove the current (x_i, time delay$_i$) from *messagebag*

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$” “EnvIn”} with
X_M = {(“CtrlIn$_1$”, V_{ctrl}), (“CtrlIn$_2$”, V_{ctrl}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”} with
Y_M = {(“CtrlOut$_1$”, $Y_{CtrlOut1}$), (“CtrlOut$_2$”, $Y_{CtrlOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, lastCDPower*} = {{“passive”, “respond”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”} x *V*}

External Transition Function:

```
$\delta_{ext}$(phase, $\sigma$, store, temperature, overtemp, overpower, lastCDPower, e, (($p_i$,$v_i$),…. ($p_n$,$v_n$))) =
(“respond”, 0, store, temperature, overtemp, overpower, lastCDPower)
  if phase = “passive” and p = “CtrlIn$_1$”
    ctrlOutput = ctrlMsg(store)
    if ctrlMsg.status = “init” or “get status”
      outputPort = “CtrlOut$_1$”
    if ctrlMsg.status = “fire laser”
      outputPort = “CtrlOut$_2$”

(“passive”, 0, store, temperature, overtemp, overpower, lastCDPower)
  if phase = “passive” and p = “CtrlIn$_2$”
     lastCDPower = messagebag.magnitude

(“passive”, $\infty$, store, temperature, overtemp, overpower, lastCDPower)
   if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
```

overtemp = "Y"

(*phase*, *σ – e*, *store, temperature*, *overtemp*, *overpower, lastCDPower*)
otherwise;

Internal Transition Function:

δ_{int}(*phase*, *σ, store, temperature*, *overtemp*, *overpower, lastCDPower*) =
("passive", ∞, *store, temperature*, *overtemp*, *overpower, lastCDPower*)
if *phase* = "respond"

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:
λ(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, lastCDPower*) =
(*outputPort*, *ctrlOutput*)
if phase = "respond"

∅ (null output)
otherwise;

Time advance Function:
ta(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower, lastCDPower*) = *σ*;

U.8 Classical Pulse Generator (CPG) Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

For the CPG compound module we define:

Parallel-DEVS *compound N*= (*X, Y, D, {*M_d | d ∈ *D*}, *EIC*, *EOC*, *IC*)

Where:

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid \text{p} \in OPorts,\ v \in Y_p\}$ is the set of output ports and values;
D = set of component names;
$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ is a DEVS atomic model;
$X_d = \{(p,v) \mid \text{p} \in IPorts,\ v \in X_p\}$;
$Y_d = \{(p,v) \mid \text{p} \in OPorts,\ v \in Y_p\}$;
$EIC \subseteq \{((N, ip_N),(d,ip_d)) \mid ip_N \in IPorts,\ d \in D,\ ip_d \in Iports_d\}$;
$EOC \subseteq \{((d,op_d),(N,op_N)) \mid op_N \in OPorts,\ d \in D,\ op_d \in Oports_d\}$;
$IC \subseteq \{((a,op_a),(b,ip_b)) \mid a,b \in D,\ op_a \in Oports_a,\ ip_b \in Iports_b\}$;
$((d,op_d),(e,ip_d)) \in IC$ implies $d \neq e$ (no feedback loops);
M = an atomic instance of P-DEVS.
N = a compound instance of P-DEVS.

$DEVS_{CPG} = (X, Y, D, \{M_d \mid \text{d} \in D\}, EIC, EOC, IC)$

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$”, “OptIn$_1$”, “OptIn$_3$”, “OptIn$_4$”, “EnvIn”}
X = {(“CtrlIn$_1$”, v), (“CtrlIn$_2$”, v), (“OptIn$_1$”, v), (“OptIn$_3$”, v), (“OptIn$_4$”, v), (“EnvIn”, v) |$v \in V$}

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”, “OptOut$_1$”, “OptOut$_3$”, “OptOut$_4$”}
Y = {(“CtrlOut$_1$”, v), (“CtrlOut$_2$”, v), (“OptOut$_1$”, v), (“OptOut$_3$”, v), (“OptOut$_4$”, v) |$v \in V$}

D = {controller, laser, isolator, polarizer, bandpass, beamsplitter, classicaldetector, PMfiber$_1$, PMfiber$_2$, PMfiber$_3$, PMfiber$_4$, PMfiber$_5$, PMfiber$_6$}
$M_d = M_{controller}, M_{laser}, M_{isolator}, M_{polarizer}, M_{bandpass}, M_{beamsplitter}, M_{classicaldetector}, M_{PMfibe1r}, M_{PMfiber2}, M_{PMfiber3}, M_{PMfiber4}, M_{PMfiber5}, M_{PMfiber6}$

EIC = {((N, “CtrlIn$_1$”),(controller, “CtrlIn$_1$”)), ((N, “EnvIn”),(controller, “EnvIn”)), ((N, “EnvIn”),(laser, “EnvIn”)), ((N, “EnvIn”),(isolator, “EnvIn”)), ((N, “EnvIn”),(polarizer, “EnvIn”)), ((N, “EnvIn”),(bandpass, “EnvIn”)), ((N, “EnvIn”),(beamsplitter, “EnvIn”)), ((N, “EnvIn”),(classicaldetector, “EnvIn”)), ((N, “EnvIn”),(PMfiber$_1$, “EnvIn”)), ((N, “EnvIn”),(PMfiber$_2$, “EnvIn”)), ((N, “EnvIn”),(PMfiber$_3$, “EnvIn”)), ((N, “EnvIn”),(PMfiber$_4$, “EnvIn”)), ((N, “EnvIn”),(PMfiber$_5$, “EnvIn”)), ((N, “EnvIn”),(PMfiber$_6$, “EnvIn”)), ((N, “OptIn$_1$”),(PMfiber$_5$, “OptIn$_2$”))}

EOC = {((PMfiber$_5$, “OptOut$_2$”),(N, “OptOut$_1$”)), ((controller, “CtrlOut$_1$”),(N, “CtrlOut$_1$”))}

IC = {((controller, “CtrlOut$_2$”), (laser, “CtrlIn”)), ((classicaldetector, “CtrlOut$_1$”),(controller, “CtrlIn$_2$”)), ((laser, “OptOut$_1$”),(PMfiber$_1$, “OptIn$_1$”)), ((PMfiber$_1$, “OptOut$_1$”),(laser, “OptIn$_1$”)), ((PMfiber$_1$, “OptOut$_2$”), (isolator, “OptIn$_1$”)), ((isolator, “OptOut$_1$”), (PMfiber$_1$, “OptIn$_2$”)), ((isolator “OptOut$_2$”), (PMfiber$_2$, “OptIn$_1$”)), ((PMfiber$_2$, “OptOut$_1$”), (isolator, “OptIn$_2$”)), ((PMfiber$_2$, “OptOut$_2$”), (polarizer, “OptIn$_1$”)), ((polarizer, “OptOut$_1$”), (PMfiber$_2$, “OptIn$_2$”)), ((polarizer, “OptOut$_2$”), (PMfiber$_3$, “OptIn$_1$”)), ((PMfiber$_3$, “OptOut$_1$”), (polarizer, “OptIn$_2$”)), ((PMfiber$_3$, “OptOut$_2$”), (bandpass, “OptIn$_1$”)), ((bandpass, “OptOut$_1$”), (PMfiber$_3$, “OptIn$_2$”)), ((bandpass “OptOut$_2$”), (PMfiber$_4$, “OptIn$_1$”)), ((PMfiber$_4$, “OptOut$_1$”), (bandpass, “OptIn$_2$”)), ((PMfiber$_4$, “OptOut$_2$”), (beamsplitter, “OptIn$_1$”)), ((beamsplitter, “OptOut$_1$”), (PMfiber$_4$, “OptIn$_2$”)), ((beamsplitter, “OptOut$_4$”), (PMfiber$_5$, “OptIn$_1$”)), ((PMfiber$_5$, “OptOut$_1$”), (beamsplitter, “OptIn$_4$”)), ((beamsplitter, “OptOut$_3$”),(PMfiber$_6$, “OptIn$_2$”)), ((PMfiber$_6$, “OptOut$_2$”),(beamsplitter, “OptIn$_3$”)), ((PMfiber$_6$, “OptOut$_1$”),(classicaldetector, “OptIn$_1$”)), ((classicaldetector, “OptOut$_1$”),(PMfiber$_6$, “OptIn$_1$”))}

U.9 CPG Controller Use Cases

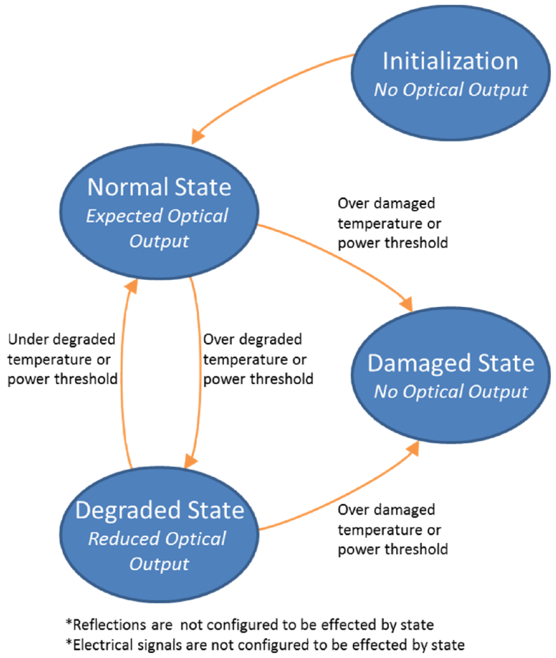

Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold
*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 206. Component states.

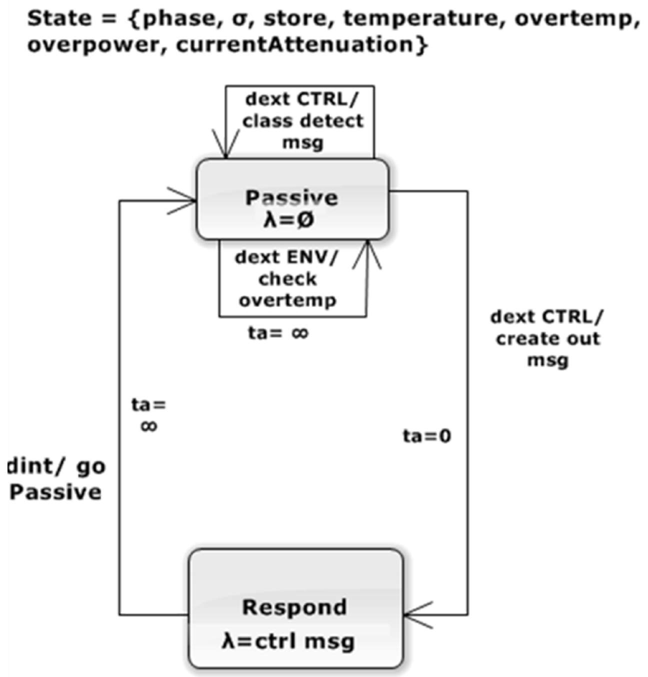

State = {phase, σ, store, temperature, overtemp, overpower, currentAttenuation}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 207. Controller phase transition diagram

U.9.1 Respond to a Reset Message

Incoming reset message arrives at the module from the quantum controller. Pass the message to the module controller. Controller clears any stored variable values and prepares an acknowledgement message. Response message is sent out the appropriate port.

- Identified Alternative Uses Cases
  - React to an environmental message
  - React to a status request message
  - React to a fire laser message
  - React to a classical detector pulse detection message

- Assumptions
  - Incoming electrical signals are not affected by component state

U.9.2 Respond to Reset Message End Goals

- Message properly received
- Controller enters Respond phase and sets storage values to zero.
- Controller forwards Reset Message to proper component(s) as necessary
- Acknowledgement message created and sent out the appropriate port
- Controller ends in Passive phase

U.9.3 Respond to an Environmental Packet

Environmental packet arrives at the controller. Check to see if environmental packet temperature sets the controller to degraded or damaged state. Check to see if temperature level returns controller from degraded state to normal state. Records change in condition, if applicable. Change controller function if in degraded or damaged state, if necessary.

- Assumptions
  - None

U.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

U.9.5 Respond to a Status Request Message

Status Request message arrives at the module from the quantum controller. Module controller prepares response message. Response message is sent out the appropriate port.

- Assumptions
    - Controller has completed initialization sequence at least once

U.9.6 Respond to Status Request End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

U.9.7 Respond to a Fire Laser Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to laser component.

- Assumptions
    - Controller has completed initialization sequence at least once

U.9.8 Respond to Fire Laser Message End Goals

- Fire laser message received properly
- Fire message recognized and passed to laser

U.9.9 Respond to a Classical Detection Message

Incoming detection message arrives at the controller from the classical detector. Store the message contents.

- Assumptions
    - Controller has completed initialization sequence at least once

U.9.10 Respond to Classical Detection Message End Goals

- CD message received properly
- CD message values stored properly

U.10 CPG Module Use Cases

U.10.1 Respond to an Optical Packet

Optical packet arrives at the module. Pass the optical packet to the proper internal component.

- Assumptions
    - Reflections are not affected by module or component state

U.10.2 Respond to Optical Packet End Goals

- Optical packet sent to proper internal component

U.10.3 Respond to an Environmental Message

Environmental packet arrives at the module. Environmental message is passed to the module controller and each component in the module.

- Assumptions
    - Incoming electrical signals are not affected by component state

U.10.4 Respond to Environmental Message End Goals

- Environmental packet received properly and forwarded to each component

U.10.5 Respond to a Control Message

Control message arrives at the module. Control message is passed to the module controller.

- Assumptions
    - Incoming electrical signals are not affected by component state

U.10.6 Respond to Environmental Message End Goals

- Control message received properly and forwarded to the module controller

U.11 CPG Test Cases

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The type of control message passed to each coupled submodule depended on the internal components.

- CPG submodule – control message fires the signal laser

These test cases led to iterations of testing and correction. Optical messages were tracked through the internal components and out the submodule output. Environmental messages were checked to ensure they replicated to each internal component. All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

	total tests	optical ports	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

U.12 References

OZOptics. (2013). Beam splitters/combiners. Retrieved, 2013, Retrieved from http://www.ozoptics.com/ALLNEW_PDF/DTS0095.pdf

OZOptics. (2014). Accurate alignment preserves polarization. Retrieved, 2014, Retrieved from http://www.ozoptics.com/ALLNEW_PDF/ART0001.pdf

RPPhotonics. (2013). Polarization-maintaining fibers. Retrieved, 2014, Retrieved from http://www.rp-photonics.com/polarization_maintaining_fibers.html

Saleh, B. E. A., & Teich, M. C. (1991). Laser diodes. *Fundamentals of photonics* (2nd ed., pp. 716-717). New York: John Wiley & Sons, Inc.

ThorLabs. (2013a). Bandpass filter structure. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1000

ThorLabs. (2013b). Coherent sources. Retrieved, 2013, Retrieved from http://www.thorlabs.com/navigation.cfm?guide_id=31

ThorLabs. (2013c). In-line fiber-optic polarizers. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=5922

ThorLabs. (2013d). Optical isolator tutorial. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6178

ThorLabs. (2013e). Photodiode tutorial. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=2822

.

Appendix V - Pulse Modulator (PM)

V.1 Device Description:

The pulse modulator creates the polarization encoding necessary for the BB84 protocol. This subsystem reacts to commands from the quantum controller to polarize the optical pulses generated by the CPG laser. Some QKD systems use a laser for each type of polarization, but this architecture uses one laser and component to change each packet polarization. The PM subsystem contains the components shown in Fig. 1.

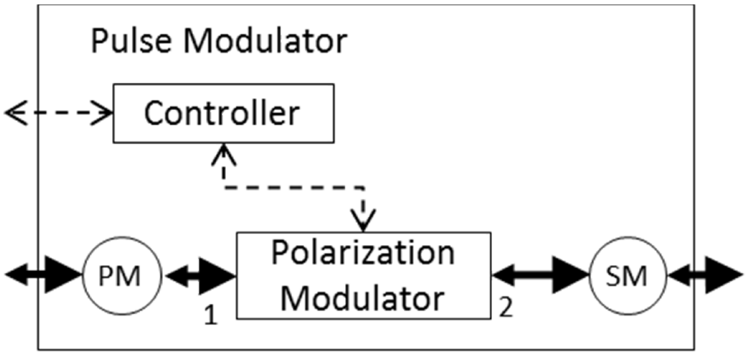


Figure 208. Pulse Modulator (PM) in the QKD system architecture.

The PM subsystem contains a controller, a polarization modulator, electrical interfaces, and interconnecting single-mode (SM) and polarization-controlling optical fiber. We briefly discuss the behavior of each of the components contained within the module.

V.1.1 PM Controller

The controller is an electrical device containing digital and analog circuits responsible for controlling the module. It has a bidirectional electrical interface to the quantum module controller and an electrical output to the controlled device. It receives commands from the quantum model controller and sends control messages to and from the controlled device.

V.1.2 Polarization Modulator

The polarization modulator (PM) is an abstract component that represents any number of devices used to electronically change the polarization of the light stream. This architecture conceptualizes these devices as having some form of polarization material that can be moved. The effect is to change a known polarization to into one of several output polarizations. The PM responds to external commands to set the output polarization to a fixed level and cannot determine the input polarization. The PM is an in-line bidirectional optical component with two optical ports. Optical signals arriving at one of the ports is attenuated and polarized, then propagated to the other port after a defined propagation delay. The optical output no longer needs the PM fiber, so the output path changes to single-mode (SM) optical fiber, which couples to the input of the next subsystem.

V.1.3 Single-Mode (SM) Optical Fiber

SM fiber is an optical component used to interconnect optical devices. It has two bidirectional optical ports. Optical signals arriving at one port propagate to the other port after a defined propagation delay with its attenuation a function of the type and the length of the fiber. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass. It has a core which guides the light and an outer cladding that reflects the internal light back into the core, bouncing the light down the fiber. This cladding helps to reflect outside light to keep in from entering the core. This structure allows for low loss over long distances. The single-mode of the fiber comes from using a small core diameter (~10μm @ 1550nm) and small numerical aperture with the fundamental mode having a bell-shaped spatial distribution similar (Saleh & Teich, 1991; ThorLabs, 2013).

V.2 PM and Controller Behavior

The controller and individual components are sensitive to the temperature in the environment in which they operate. If the temperature exceeds defined thresholds, the components may become temporarily degraded or permanently damaged which changes their characteristics. If temporarily degraded, the devices may recover to normal operating behavior after the temperature returns to a "normal" operating temperature.

The first step involved with modeling the controller and module is to collect and understand the physical, behavioral, and performance characteristics of the atomic components. In this case, the individual components were constructed earlier and the controller was built as a message handler. The logic for the controller was based on the types of messages necessary for control of components inside the module.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

V.3 PM Compound Conceptual Model

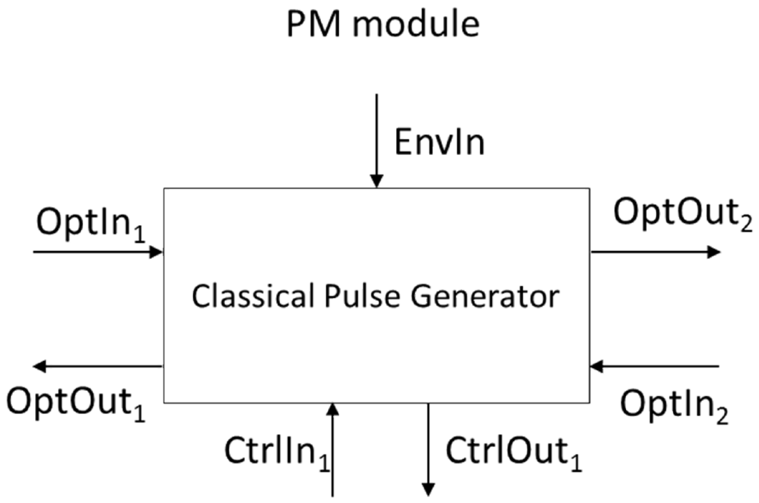


Figure 209. Pulse Modulator (PM) compound module conceptual model.

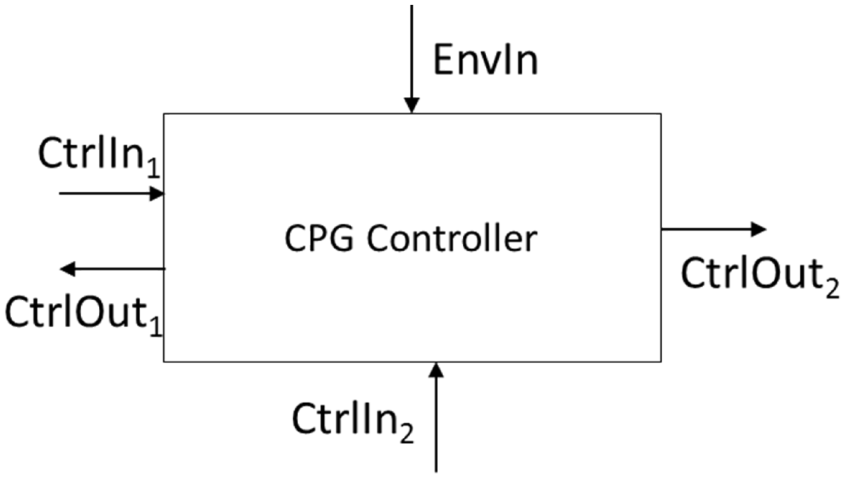


Figure 210. Pulse Modulator (PM) controller conceptual model

Table 86. *List of PM Controller messages.*

Input Messages	**From**	**Response**
PM_ENV	Quantum controller	Set the internal controller temperature
PM_RESET	Quantum controller	Resets the controller and clears the state variables
PM_STATUS_REQUEST	Quantum controller	Sends the controller status
PM_SET_HORIZONTA	Quantum	Sets polarization to horizontal

L	controller	
PM_SET_VERTICAL	Quantum controller	Sets polarization to vertical
PM_SET_ANTIDIAGONAL	Quantum controller	Sets polarization to anti-diagonal
PM_SET_DIAGONAL	Quantum controller	Sets polarization to diagonal
PM_SET_ANGLE	Quantum controller	Sets polarization to a specific angle
PM_GET_ANGLE	Quantum controller	Requests current polarization angle of the polarization modulator
Output Messages	**To**	**Content**
PM_ACK	Quantum Controller	Response to a Reset message
PM_STATUS	Quantum Controller	Response to a Status Request message

The conceptual model for a PM consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_1$}, one environmental input port {EvnIn}, one control input port {CtrlIn$_1$} and one control output port {CtrlOut$_1$}. The environmental port allows external sources to communicate changes in the operational environment to the module. The electrical controller ports allow for control inputs to the controller and responses from the module to the higher system functions.

In comparison to the module layout used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism. The electrical control port is also decomposed in the model into an input port and an output port.

When an optical signal is sent to the input of the module, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The module components must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters each of the components the module. In the case of optical overpowering, once overpowered a component will permanently change attenuation. External environmental messages sent to the module are directed to individual components to convey the temperature of the operational environmental so the module can determine if it is degraded (a temporary condition) or damaged (a permanent condition). Changes to components based on the temperature determine the behavior of the module.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

V.4 English-Language Rules for the Controller

In this section, English language rules are developed to express the desired behavior of the controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When a control signal arrives:

- Determine the arrival port of the signal.

- Evaluate the content of the message
- Generate a response message to the incoming signal (if necessary).
- Generate a forwarded message to the appropriate device (if necessary).
- Output the response or forwarded message out the appropriate port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

V.5 DEVS Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the module controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs.

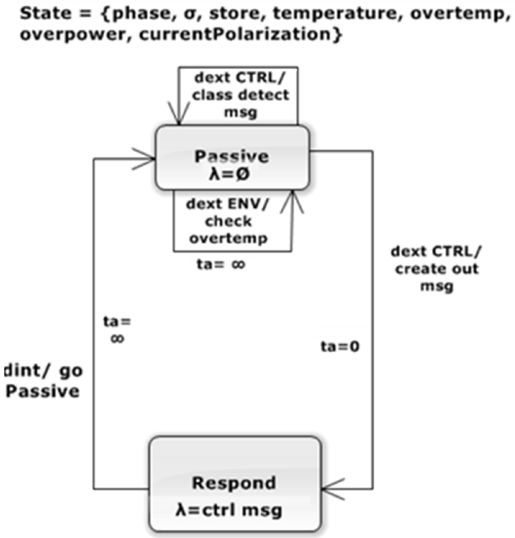

State = {phase, σ, store, temperature, overtemp, overpower, currentPolarization}
dext CTRL/ class detect msg
Passive λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond λ=ctrl msg

Figure 211. PM Controller DEVS phase transition diagram

V.6 PM Controller Event-Trace Diagram

This section shows various examples of messages entering the controller. The tables list the states the component proceeds through as the events are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the component, the over temperature flag variable and the over power flag variable. The queue column shows the contents of the queue at that state, the contents of the store state variable and any notes. Note in contrast to most other components, the controller is very simple and only responds to incoming messages; it does not generate any messages on its own. There are two types of inputs: control messages and environmental messages.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- CurrentPolarization: current polarization modulator polarization setting
- Notes: any notes for that state

V.6.1 CASE I: Initial Passive with Single Control Packet Arriving at Time 0

Table 87. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**current polari- zation**	**Notes: assume tp=0**
	1-packet	no env	no ext	1 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	

0	s0	exit	passive	0	null	c	n	n	null	
0	s1	entry	respond	0	null	c	n	n	null	
0	s1	exit	respond	inf	null	c	n	n	null	
0	s2	entry	passive	inf	null	c	n	n	null	

V.6.2 CASE II: Initial Passive with Single Environmental Packet Arriving at Time 0

Table 88. *Case II state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	current polari- zation	Notes: assume tp=5
	1-packet	1 env	no ext	0 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	S1	entry	passive	inf	null	c	n	n	null	

V.7 PM Controller Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “currentPolarization”}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
“currentPolarization” = variable to store the current polarization value of the polarization modulator

For the controller we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values;

$Y_M = \{(p,v) \mid \text{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

$DEVS_{PMcontroller} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

where

t_p = transmission time inside the component
temperature = current temperature of the component
phase = control state that keeps track of the internal phase of the component
phase = {“passive”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
currentPolarization = variable that holds the current polarization
interruptRespond = flag variable set when Respond phase is interrupted by an external event
messagebag= variable that stores the current x input value(s) (p,v)
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
ctrlOutput = variable that stores the output control message response
output.port = variable that holds the output optical packet port
store = variable that holds values of the current input values
timeLeftRespond = time left in Respond phase for the current event
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
ctrlMsg() = method that generates a response message to received control messages
messagebag_first() = method that returns the first element of the message bag
remove_event_m() = method that remove the current (x_i, time delay$_i$) from *messagebag*

Every δ_{ext} puts all of its x (p,v) values into the variable *store*

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$” “EnvIn”} with
X_M = {(“CtrlIn$_1$”, V_{ctrl}), (“CtrlIn$_2$”, V_{ctrl}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”} with

$Y_M = \{(\text{“CtrlOut}_1\text{”}, Y_{CtrlOut1}), (\text{“CtrlOut}_2\text{”}, Y_{CtrlOut2})\}$ is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

$S = \{phase, \sigma, store, temperature, overtemp, overpower, currentPolarization\} = \{\{\text{“passive”}, \text{“respond”}\} \text{ x } R_0^+ \text{ x } V \text{ x } R \text{ x } \{\text{“Y”}, \text{“N”}\} \text{ x } \{\text{“Y”},\text{”N”}\} \text{ x } V\}$

External Transition Function:

```
$\delta_{ext}(phase, \sigma, store, temperature, overtemp, overpower, currentPolarization, e, ((p_i,v_i),\ldots(p_n,v_n))) =$
$(\text{“respond”}, 0, store, temperature, overtemp, overpower, currentPolarization)$
  if $phase$ = “passive” and $p$ = “CtrlIn$_1$”
    $ctrlOutput$ = ctrlMsg($store$)
    if $ctrlMsg.status$ = “init” or “get status” or “get angle”
      $outputPort$ = “CtrlOut$_1$”
    if $ctrlMsg.status$ = “any set msg”
      $outputPort$ = “CtrlOut$_2$”

$(\text{“passive”}, 0, store, temperature, overtemp, overpower, currentPolarization)$
  if $phase$ = “passive” and $p$ = “CtrlIn$_2$”
    $currentPolarization = messagebag.polarization$

$(\text{“passive”}, \infty, store, temperature, overtemp, overpower, currentPolarization)$
  if $phase$ = “passive” and $p$ = “EnvIn”
    $temperature = messagebag.temperature$
    if $temperature > damage.temp$
      $overtemp$ = “Y”

$(phase, \sigma - e, store, temperature, overtemp, overpower, currentPolarization)$
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}(phase, \sigma, store, temperature, overtemp, overpower, currentPolarization) =$
  $(\text{“passive”}, \infty, store, temperature, overtemp, overpower, currentPolarization)$
    if $phase$ = “respond”
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:

```
$\lambda(phase, \sigma, store, temperature, overtemp, overpower, currentPolarization) =$
  $(outputPort, ctrlOutput)$
    if phase = “respond”
```

∅ (null output)
otherwise;

Time advance Function:
$ta(phase, \sigma, store, temperature, overtemp, overpower, currentPolarization) = \sigma$;

V.8 PM Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

For the PM compound module we define:

Parallel-DEVS *compound* $N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$

Where:

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;
D = set of component names;
$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ is a DEVS atomic model;
$X_d = \{(p,v) \mid p \in IPorts, v \in X_p\}$;
$Y_d = \{(p,v) \mid p \in OPorts, v \in Y_p\}$;
$EIC \subseteq \{((N, ip_N),(d,ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in Iports_d\}$;
$EOC \subseteq \{((d,op_d),(N,op_N)) \mid op_N \in OPorts, d \in D, op_d \in Oports_d\}$;
$IC \subseteq \{((a,op_a),(b,ip_b)) \mid a,b \in D, op_a \in Oports_a, ip_b \in Iports_b\}$;
$((d,op_d),(e,ip_d)) \in IC$ implies $d \neq e$ (no feedback loops);
M = an atomic instance of P-DEVS.
N = a compound instance of P-DEVS.

$$\boldsymbol{DEVS_{PM} = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)}$$

InPorts = {“CtrlIn_1”, “CtrlIn_2”, “OptIn_1”, “OptIn_2”, “EnvIn”}
X = {(“CtrlIn_1”, v), (“CtrlIn_2”, v), (“OptIn_1”, v), (“OptIn_2”, v), (“EnvIn”, v) $|v \in V$}

OutPorts = {“CtrlOut_1”, “CtrlOut_2”, “OptOut_1”, “OptOut_2”}

Y = {(“CtrlOut_1”, v), (“CtrlOut_2”, v), (“OptOut_1”, v), (“OptOut_2”, v)|$v \in V$}

D = {controller, polarizationmodulator, PMfiber, SMfiber}
$M_d = M_{controller}, M_{polarizationmodulator}, M_{PMfiber}, M_{SMfiber}$

EIC = {((N, “CtrlIn_1”),(controller, “CtrlIn_1”)), ((N, “EnvIn”),(controller, “EnvIn”)), ((N, “EnvIn”),(polarizationmodulator, “EnvIn”)), ((N, “EnvIn”),(PMfiber, “EnvIn”)), ((N, “EnvIn”),(SMfiber, “EnvIn”)),((N, “OptIn_1”),(PMfiber, “OptIn_1”)), ((N, “OptIn_2”),(SMfiber, “OptIn_2”))}

EOC = {((PMfiber, “OptOut_1”),(N, “OptOut_1”)), ((controller, “CtrlOut_1”),(N, “CtrlOut_1”)), ((SMfiber, “OptOut_2”),(N, “OptOut_2”))}

IC = {((controller, “CtrlOut_2”), (polarizationmodulator, “CtrlIn_1”)), ((polarization modulator, “CtrlOut_1”),(controller, “CtrlIn_2”)) ,((PMfiber, “OptOut_2”), (polarizationmodulator, “OptIn_1”)), ((polarizationmodulator, “OptOut_1”), (PMfiber, “OptIn_2”)), ((polarizationmodulator, “OptOut_2”), (SMfiber, “OptIn_1”)), ((SMfiber, “OptOut_1”), (polarizationmodulator, “OptIn_2”))}

V.9 PM Controller Use Cases

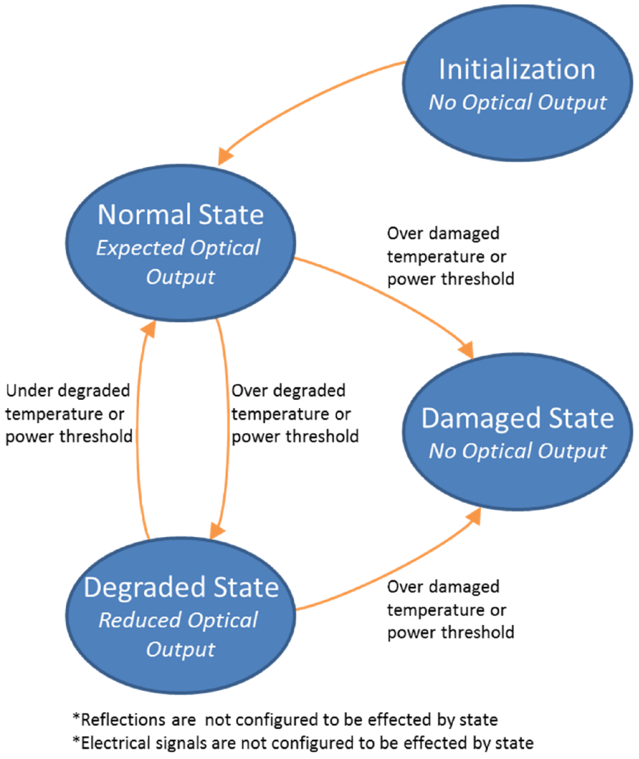

Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold
*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 212. Component states.

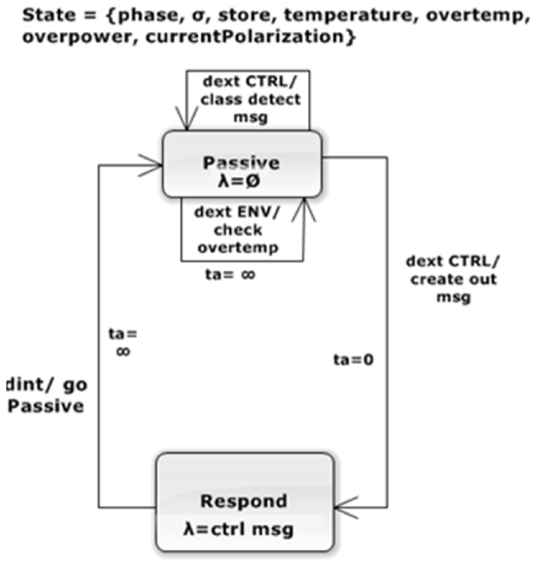

State = {phase, σ, store, temperature, overtemp, overpower, currentPolarization}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 213. Controller phase transition diagram

V.9.1 Respond to a Reset Message

Incoming reset message arrives at the module from the quantum controller. Pass the message to the module controller. Controller clears any stored variable values and prepares an acknowledgement message. Response message is sent out the appropriate port.

- Identified Alternative Uses Cases
  - React to an environmental message
  - React to a status request message
  - React to a set horizontal message
  - React to a set vertical message
  - React to a set antidiagonal message
  - React to a set diagonal message
  - React to a set angle message
  - React to a get angle message

- Assumptions
  - Incoming electrical signals are not affected by component state

V.9.2 Respond to Reset Message End Goals

- Message properly received
- Controller enters Respond phase and sets storage values to zero.
- Controller forwards Reset Message to proper component(s) as necessary
- Acknowledgement message created and sent out the appropriate port

- Controller ends in Passive phase

V.9.3 Respond to an Environmental Packet

Environmental packet arrives at the controller. Check to see if environmental packet temperature sets the controller to degraded or damaged state. Check to see if temperature level returns controller from degraded state to normal state. Records change in condition, if applicable. Change controller function if in degraded or damaged state, if necessary.

- Assumptions
    - None

V.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

V.9.5 Respond to a Status Request Message

Status Request message arrives at the module from the quantum controller. Module controller prepares response message. Response message is sent out the appropriate port.

- Assumptions
    - Controller has completed initialization sequence at least once

V.9.6 Respond to Status Request End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

V.9.7 Respond to a Set Horizontal Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
    - Controller has completed initialization sequence at least once

V.9.8 Respond to Set Horizontal Message End Goals

- Set Horizontal message received properly
- Message recognized and passed to the proper component

V.9.9 Respond to a Set Vertical Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

V.9.10 Respond to Set Vertical Message End Goals

- Set Vertical message received properly
- Message recognized and passed to the proper component

V.9.11 Respond to a Set AntiDiagonal Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

V.9.12 Respond to Set AntiDiagonal Message End Goals

- Set AntiDiagonal message received properly
- Message recognized and passed to the proper component

V.9.13 Respond to a Set Diagonal Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

V.9.14 Respond to Set Diagonal Message End Goals

- Set Diagonal message received properly

- Message recognized and passed to polarization controller

V.9.15 Respond to a Set Angle Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to proper component

. Assumptions

  - Controller has completed initialization sequence at least once

V.9.16 Respond to Set Angle Message End Goals

- Set Angle message received properly
- Message recognized and passed to proper component

V.9.17 Respond to a Get Angle Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

V.9.18 Respond to Get Angle Message End Goals

- Get Angle message received properly
- Message recognized and passed to the proper component

V.10 PM Module Use Cases

V.10.1 Respond to an Optical Packet

Optical packet arrives at the module. Pass the optical packet to the proper internal component.

- Assumptions
  - Reflections are not affected by module or component state

V.10.2 Respond to Optical Packet End Goals

- Optical packet sent to proper internal component

V.10.3 Respond to an Environmental Message

Environmental packet arrives at the module. Environmental message is passed to the module controller and each component in the module.

- Assumptions
    - Incoming electrical signals are not affected by component state

V.10.4 Respond to Environmental Message End Goals

- Environmental packet received properly and forwarded to each component

V.10.5 Respond to a Control Message

Control message arrives at the module. Control message is passed to the module controller.

- Assumptions
    - Incoming electrical signals are not affected by component state

V.10.6 Respond to Environmental Message End Goals

- Control message received properly and forwarded to the module controller

V.11 PM Test Cases

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The type of control message passed to each coupled submodule depended on the internal components.

- PM submodule – control message changes polarization of polarization controller

These test cases led to iterations of testing and correction. Optical messages were tracked through the internal components and out the submodule output. Environmental messages were checked to ensure they replicated to each internal component. All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

	total tests	optical ports	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

V.12 References

Saleh, B. E. A., & Teich, M. C. (1991). Guided waves. *Fundamentals of photonics* (2nd ed., pp. 340-342). New York: John Wiley & Sons, Inc.

ThorLabs. (2013). Single-mode fiber. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=949

Appendix W - Decoy State Generator (DSG)

W.1 Device Description:

The ideal version of a QKD system would emit single photons, but existing hardware is not capable of producing on-demand single photons. This allows eavesdroppers the opportunity to conduct a ‘photon-number-splitting attack’ (PNS) but Alice and Bob have countermeasures in ‘decoy states.’ The Decoy State Generator (DSG) allows the quantum controller to randomly vary the power of the optical pulses. This variance, along with statistic collection, allows Alice and Bob to detect PNS activity in the quantum channel. The DSG contains the components shown in Fig. 1.

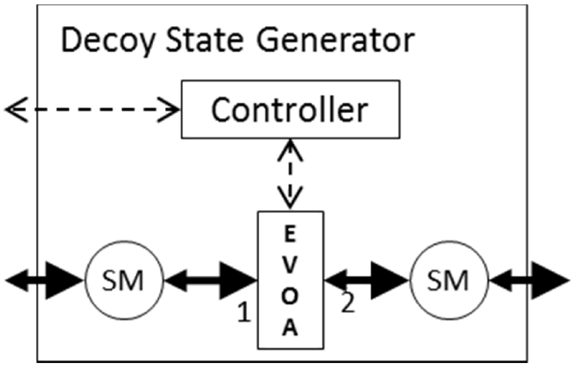


Figure 214. Decoy State Generator (DSG) in the QKD system architecture.

The DSG subsystem contains a controller, an electronically variable optical attenuator (EVOA), electrical interfaces, and interconnecting single-mode (SM) optical fiber. We briefly discuss the behavior of each of the components contained within the module.

W.1.1 DSG Controller

The controller is an electrical device containing digital and analog circuits responsible for controlling the EVOA. It has a bidirectional electrical interface to the quantum module controller and an electrical output to the EVOA. It receives commands from the quantum model controller to vary the attenuation, creating differing power levels within the optical pulses.

W.1.2 EVOA

The EVOA is an opto-electrical device containing a variable attenuator and support electronics to vary the output attenuation. The EVOA attenuates the power of optical signals by a variable amount. These devices usually have some form of blocking material such as an opaque slab or a window tilted in the path of the light. This blocking material is connected to an electric motor controlled by the higher system functions, allowing for a variable amount of light to exit the device (OZOptics, 2013; ThorLabs, 2013b). Optical signals arriving at one port propagate to the other port after a defined propagation delay with the attenuation based on the current controlled value. The EVOA output couples to input of the next subsystem via SM fiber.

W.1.3 Single-Mode Optical Fiber

SM fiber is an optical component used to interconnect optical devices. It has two bidirectional optical ports. Optical signals arriving at one port propagate to the other port after a defined propagation delay with its attenuation a function of the type and the length of the fiber. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass. It has a core which guides the light and an outer cladding that reflects the internal light back into the core, bouncing the light down the fiber. This cladding helps to reflect outside light to keep in from entering the core. This structure allows for low loss over long distances. The single-mode of the fiber comes from using a small core diameter (~10µm @ 1550nm) and small numerical aperture with the fundamental mode having a bell-shaped spatial distribution similar (Saleh & Teich, 1991; ThorLabs, 2013). SM fiber couples the EVOA with the next subsystem.

W.2 DSG and Controller Behavior

The controller and individual components are sensitive to the temperature in the environment in which they operate. If the temperature exceeds defined thresholds, the components may become temporarily degraded or permanently damaged which changes their characteristics. If temporarily degraded, the devices may recover to normal operating behavior after the temperature returns to a "normal" operating temperature.

The first step involved with modeling the controller and module is to collect and understand the physical, behavioral, and performance characteristics of the atomic components. In this case, the individual components were constructed earlier and the controller was built as a message handler. The logic for the controller was based on the types of messages necessary for control of components inside the module.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

W.3 DSG Compound Conceptual Model

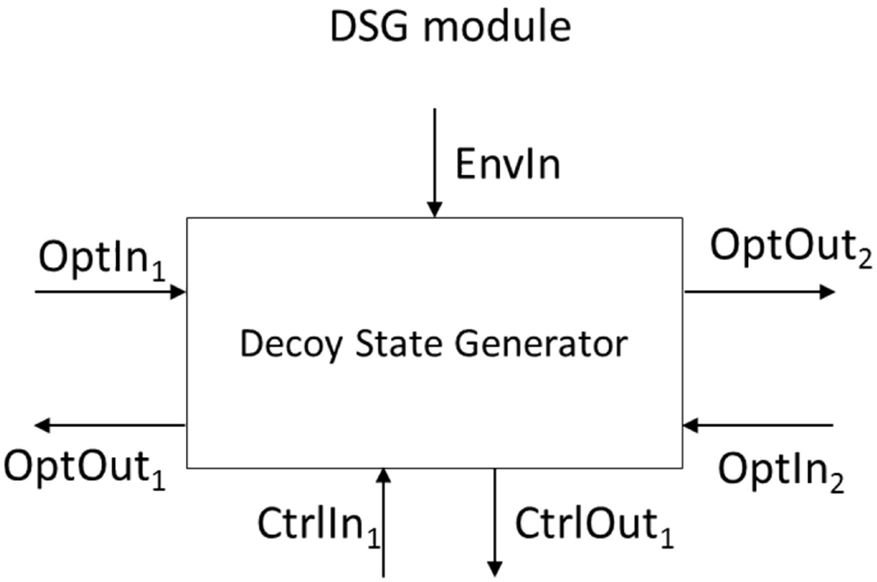


Figure 215. DSG compound module conceptual model.

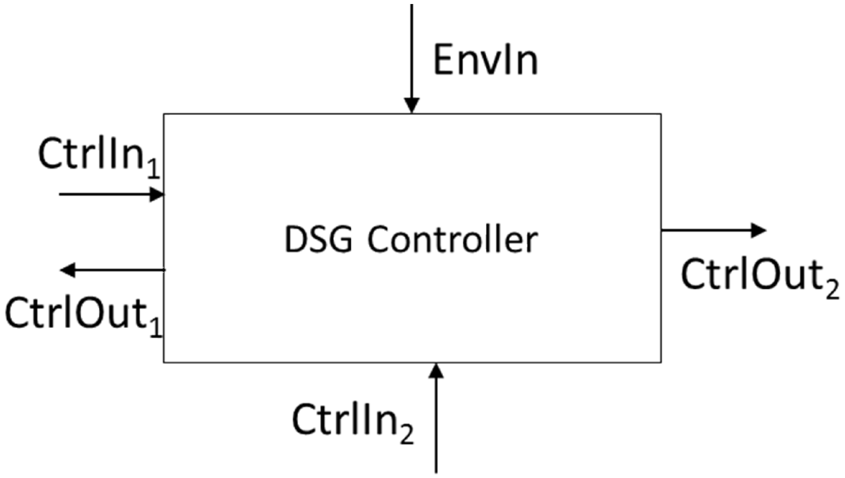


Figure 216. DSG controller conceptual model

Table 89. *List of DSG Controller messages.*

Input Messages	**From**	**Response**
DSG_ENV	Quantum controller	Set the internal controller temperature
DSG_RESET	Quantum controller	Resets the controller and clears the state variables
DSG_STATUS_REQUEST	Quantum controller	Sends the controller status
DSG_INCREASE_ATTEN	Quantum controller	Increases the attenuation
DSG_DECREASE_ATTEN	Quantum controller	Decreases the attenuation
DSG_SET_ATTEN	Quantum controller	Sets the attenuation
DSG_GET_ATTEN	Quantum controller	Gets the current attenuation value
Output Messages	**To**	**Content**
DSG_ACK	Quantum Controller	Response to a Reset message
DSG_STATUS	Quantum Controller	Response to a Status Request message

The conceptual model for a DSG consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_1$}, one environmental input port {EvnIn}, one control input port {CtrlIn$_1$} and one control output port {CtrlOut$_1$}. The environmental port allows external sources to communicate changes in the operational environment to the module. The electrical controller ports allow for control inputs to the controller and responses from the module to the higher system functions.

In comparison to the module layout used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism. The electrical control port is also decomposed in the model into an input port and an output port.

When an optical signal is sent to the input of the module, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The module components must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters each of the components the module. In the case of optical overpowering, once overpowered a component will permanently change attenuation. External environmental messages sent to the module are directed to individual components to convey the temperature of the operational environmental so the module can determine if it is degraded (a temporary condition) or damaged (a permanent condition). Changes to components based on the temperature determine the behavior of the module.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

W.4 English-Language Rules for the Controller

In this section, English language rules are developed to express the desired behavior of the controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.

- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When a control signal arrives:

- Determine the arrival port of the signal.
- Evaluate the content of the message
- Generate a response message to the incoming signal (if necessary).
- Generate a forwarded message to the appropriate device (if necessary).
- Output the response or forwarded message out the appropriate port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

W.5 DEVS Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the module controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs.

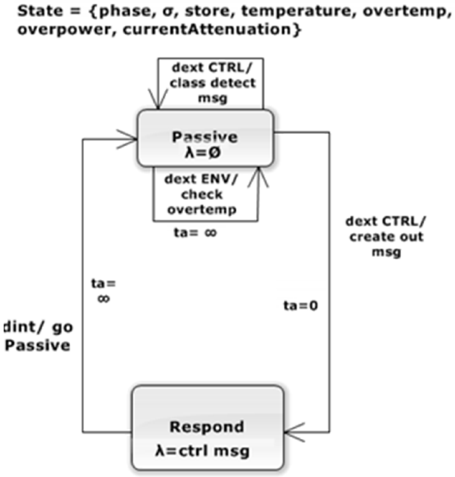

State = {phase, σ, store, temperature, overtemp, overpower, currentAttenuation}
dext CTRL/ class detect msg
Passive λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond λ=ctrl msg

Figure 217. DSG Controller DEVS phase transition diagram

W.6 DSG Controller Event-Trace Diagram

This section shows various examples of messages entering the controller. The tables list the states the component proceeds through as the events are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the component, the over temperature flag variable and the over power flag variable. The queue column shows the contents of the queue at that state, the contents of the store state variable and any notes. Note in contrast to most other components, the controller is very simple and only responds to incoming messages; it does not generate any messages on its own. There are two types of inputs: control messages and environmental messages.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable

- Over Power: shows the value of the over power flag variable
- Current Attenuation: current EVOA attenuation setting
- Notes: any notes for that state

W.6.1 CASE I: Initial Passive with Single Control Packet Arriving at Time 0

Table 90. *Case I state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**current attenuation**	**Notes: assume tp=0**
	1-packet	no env	no ext	1 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	s1	entry	respond	0	null	c	n	n	null	
0	s1	exit	respond	inf	null	c	n	n	null	
0	s2	entry	passive	inf	null	c	n	n	null	

W.6.2 CASE II: Initial Passive with Single Environmental Packet Arriving at Time 0

Table 91. *Case II state list.*

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**current attenuation**	**Notes: assume tp=5**
	1-packet	1 env	no ext	0 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	S1	entry	passive	inf	null	c	n	n	null	

W.7 DSG Controller Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

Definitions:

State = {phase, time advance, "store", temperature, "overtemp", "overpower", "currentAttenuation"}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level

“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
“currentAttenuation” = variable to store the current attenuation value of the EVOA

For the controller we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

$X_M = \{(p,v) \mid \text{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \text{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{DSGcontroller}$* **= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)**

where

t_p = transmission time inside the component
temperature = current temperature of the component
phase = control state that keeps track of the internal phase of the component
phase = {“passive”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
currentAttenuation = variable that holds the current attenuation
interruptRespond = flag variable set when Respond phase is interrupted by an external event
messagebag= variable that stores the current x input value(s) (p,v)
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
ctrlOutput = variable that stores the output control message response
output.port = variable that holds the output optical packet port
store = variable that holds values of the current input values
timeLeftRespond = time left in Respond phase for the current event
e = elapsed time since last transition occurred

σ = state variable that holds the time to next transition
ctrlMsg() = method that generates a response message to received control messages
messagebag_first() = method that returns the first element of the message bag
remove_event_m() = method that remove the current (x_i, time delay_i) from *messagebag*

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"CtrlIn_1", "CtrlIn_2" "EnvIn"} with
X_M = {("CtrlIn_1", V_{ctrl}), ("CtrlIn_2", V_{ctrl}), ("EnvIn", V_{env})} is the set of input ports and values.

OutPorts = {"CtrlOut_1", "CtrlOut_2"} with
Y_M = {("CtrlOut_1", $Y_{CtrlOut1}$), ("CtrlOut_2", $Y_{CtrlOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, currentAttenuation* } = {{"passive", "respond"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x *V* }

External Transition Function:

```
δ_ext(phase, σ, store, temperature, overtemp, overpower, currentAttenuation ,e, ((p_i,v_i),…. (p_n,v_n)))
                                                                                   =
("respond", 0, store, temperature, overtemp, overpower, currentAttenuation)
  if phase = "passive" and p = "CtrlIn_1"
    ctrlOutput = ctrlMsg(store)
    if ctrlMsg.status = "init" or "get status" or "get attenuation"
      outputPort = "CtrlOut_1"
    if ctrlMsg.status = "increase" or "decrease" or set"
      outputPort = "CtrlOut_2"

("passive", 0, store, temperature, overtemp, overpower, currentAttenuation)
  if phase = "passive" and p = "CtrlIn_2"
    currentAttenuation = messagebag.attenuation

("passive", ∞, store, temperature, overtemp, overpower, currentAttenuation)
  if phase = "passive" and p = "EnvIn"
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = "Y"

(phase, σ – e, store, temperature, overtemp, overpower, currentAttenuation)
  otherwise;
```

Internal Transition Function:

$\delta_{int}(phase, \sigma, store, temperature, overtemp, overpower, currentAttenuation) =$
  ("passive", ∞, $store, temperature, overtemp, overpower, currentAttenuation$)
    if $phase$ = "respond"

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$

Output Function:
$\lambda(phase, \sigma, store, temperature, overtemp, overpower, currentAttenuation) =$
  ($outputPort$, $ctrlOutput$)
      if phase = "respond"

  Ø (null output)
    otherwise;

Time advance Function:
$ta(phase, \sigma, store, temperature, overtemp, overpower, currentAttenuation) = \sigma;$

W.8 DSG Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

For the DSG compound module we define:

Parallel-DEVS *compound* $N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$

Where:

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;
D = set of component names;
$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ is a DEVS atomic model;
$X_d = \{(p,v) \mid p \in IPorts, v \in X_p\};$
$Y_d = \{(p,v) \mid p \in OPorts, v \in Y_p\};$
$EIC \subseteq \{((N, ip_N),(d,ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in Iports_d\};$

$EOC \subseteq \{((d,op_d),(N,op_N)) | \; op_N \in OPorts, \; d \in D, \; op_d \in Oports_d\}$;

$IC \subseteq \{((a,op_a),(b,ip_b)) | a,b \in D, \; op_a \in Oports_a, \; ip_b \in Iports_b\}$;

$((d,op_d),(e,ip_d)) \in IC$ implies $d \neq e$ (no feedback loops);

M = an atomic instance of P-DEVS.

N = a compound instance of P-DEVS.

$$\boldsymbol{DEVS_{DSG} = (X, Y, D, \{M_d \,|\, \mathrm{d} \in D\}, EIC, EOC, IC)}$$

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$”, “OptIn$_1$”, “OptIn$_2$”, “EnvIn”}

X = {(“CtrlIn$_1$”, *v*), (“CtrlIn$_2$”, *v*), (“OptIn$_1$”, *v*), (“OptIn$_2$”, *v*), (“EnvIn”, *v*) |*v* ∈ *V*}

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”, “OptOut$_1$”, “OptOut$_2$”}

Y = {(“CtrlOut$_1$”, *v*), (“CtrlOut$_2$”, *v*), (“OptOut$_1$”, *v*), (“OptOut$_2$”, *v*)|*v* ∈ *V*}

D = {controller, evoa, SMfiber$_1$, SMfiber$_2$}

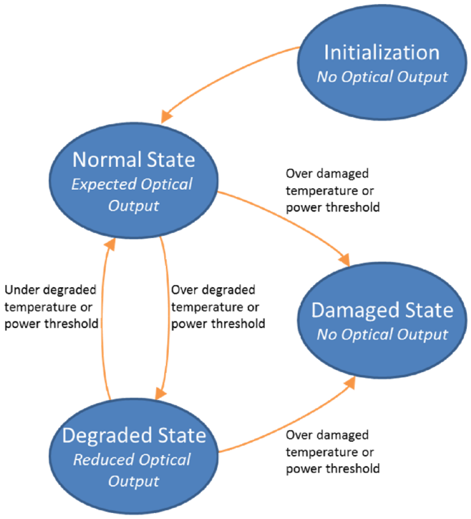
$M_d = M_{controller}, M_{evoa}, Ms_{Mfiber1}, M_{SMfiber2}$

EIC = {((*N*, “CtrlIn$_1$”),(controller, “CtrlIn$_1$”)), ((*N*, “EnvIn”),(controller, “EnvIn”)), ((*N*, “EnvIn”),(evoa, “EnvIn”)), ((*N*, “EnvIn”),(SMfiber$_1$, “EnvIn”)), ((*N*, “EnvIn”), (SMfiber$_2$, “EnvIn”)),((*N*, “OptIn$_1$”),(SMfiber$_1$, “OptIn$_1$”)), ((*N*, “OptIn$_2$”),(SMfiber$_2$, “OptIn$_2$”))}

EOC = {((SMfiber$_1$, “OptOut$_1$”),(*N*, “OptOut$_1$”)), ((controller, “CtrlOut$_1$”),(*N*, “CtrlOut$_1$”)), ((SMfiber$_2$, “OptOut$_2$”),(*N*, “OptOut$_2$”))}

IC = {((controller, “CtrlOut$_2$”), (evoa, “CtrlIn$_1$”)), ((evoa, “CtrlOut$_1$”),(controller, “CtrlIn$_2$”)) ,((SMfiber$_1$, “OptOut$_2$”), (evoa, “OptIn$_1$”)), ((evoa, “OptOut$_1$”), (SMfiber$_1$, “OptIn$_2$”)), ((evoa, “OptOut$_2$”), (SMfiber$_2$, “OptIn$_1$”)), ((SMfiber$_2$, “OptOut$_1$”), (evoa, “OptIn$_2$”))}

W.9 DSG Controller Use Cases


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 218. Component states.

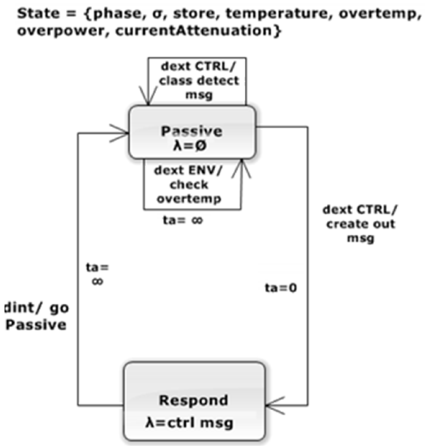

State = {phase, σ, store, temperature, overtemp, overpower, currentAttenuation}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 219. Controller phase transition diagram

W.9.1 Respond to a Reset Message

Incoming reset message arrives at the module from the quantum controller. Pass the message to the module controller. Controller clears any stored variable values and prepares an acknowledgement message. Response message is sent out the appropriate port.

- Identified Alternative Uses Cases
    - React to an environmental message
    - React to a status request message
    - React to an increase attenuation message
    - React to a decrease attenuation message
    - React to a set attenuation message

    - React to a get attenuation message
- Assumptions
    - Incoming electrical signals are not affected by component state

W.9.2 Respond to Reset Message End Goals

- Message properly received
- Controller enters Respond phase and sets storage values to zero.
- Controller forwards Reset Message to proper component(s) as necessary
- Acknowledgement message created and sent out the appropriate port
- Controller ends in Passive phase

W.9.3 Respond to an Environmental Packet

Environmental packet arrives at the controller. Check to see if environmental packet temperature sets the controller to degraded or damaged state. Check to see if temperature level returns controller from degraded state to normal state. Records change in condition, if applicable. Change controller function if in degraded or damaged state, if necessary.

- Assumptions
    - None

W.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

W.9.5 Respond to a Status Request Message

Status Request message arrives at the module from the quantum controller. Module controller prepares response message. Response message is sent out the appropriate port.

- Assumptions
    - Controller has completed initialization sequence at least once

W.9.6 Respond to Status Request End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary

- Change component function properly, if necessary

W.9.7 Respond to an Increase Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
    - Controller has completed initialization sequence at least once

W.9.8 Respond to Increase Attenuation Message End Goals

- Increase Attenuation message received properly
- Message recognized and passed to proper component

W.9.9 Respond to a Decrease Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
    - Controller has completed initialization sequence at least once

W.9.10 Respond to Decrease Attenuation Message End Goals

- Decrease Attenuation message received properly
- Message recognized and passed to proper component

W.9.11 Respond to a Set Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
    - Controller has completed initialization sequence at least once

W.9.12 Respond to Set Attenuation Message End Goals

- Set Attenuation message received properly
- Message recognized and passed to proper component

W.9.13 Respond to a Get Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
    - Controller has completed initialization sequence at least once

W.9.14 Respond to Get Attenuation Message End Goals

- Get Attenuation message received properly
- Message recognized and passed to proper component

W.10 DSG Module Use Cases

W.10.1 Respond to an Optical Packet

Optical packet arrives at the module. Pass the optical packet to the proper internal component.

- Assumptions
    - Reflections are not affected by module or component state

W.10.2 Respond to Optical Packet End Goals

- Optical packet sent to proper internal component

W.10.3 Respond to an Environmental Message

Environmental packet arrives at the module. Environmental message is passed to the module controller and each component in the module.

- Assumptions
    - Incoming electrical signals are not affected by component state

W.10.4 Respond to Environmental Message End Goals

- Environmental packet received properly and forwarded to each component

W.10.5 Respond to a Control Message

Control message arrives at the module. Control message is passed to the module controller.

- Assumptions
    - Incoming electrical signals are not affected by component state

W.10.6 Respond to Environmental Message End Goals

- Control message received properly and forwarded to the module controller

W.11 DSG Test Cases

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The type of control message passed to each coupled submodule depended on the internal components.

- DSG submodule – control message changes attenuation of EVOA

These test cases led to iterations of testing and correction. Optical messages were tracked through the internal components and out the submodule output. Environmental messages were checked to ensure they replicated to each internal component. All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

	total tests	optical ports	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

W.12 References

OZOptics. (2013). Electrically-controlled variable fiber-optic attenuator. Retrieved, 2013, Retrieved from http://www.ozoptics.com/ALLNEW_PDF/DTS0010.pdf

Saleh, B. E. A., & Teich, M. C. (1991). Guided waves. *Fundamentals of photonics* (2nd ed., pp. 340-342). New York: John Wiley & Sons, Inc.

ThorLabs. (2013a). Single-mode fiber. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=949

ThorLabs. (2013b). Variable fiber optical attenuators, single mode. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6161

Appendix X - Classical To Quantum (CTQ)

X.1 Device Description:

Optical pulses generated by the laser in the CPG contain millions of photons, far more than required by QKD protocols. Since single photon generators are not available, existing QKD systems take these classical-level pulses (meaning they contain many photons) and attenuate them down to quantum-level pulses (meaning less than one photon) with an average photon count of 0.1. This low number is necessary to achieve the single-photon requirements of QKD. The Classical To Quantum subsystem contains the components shown in Fig. 1.

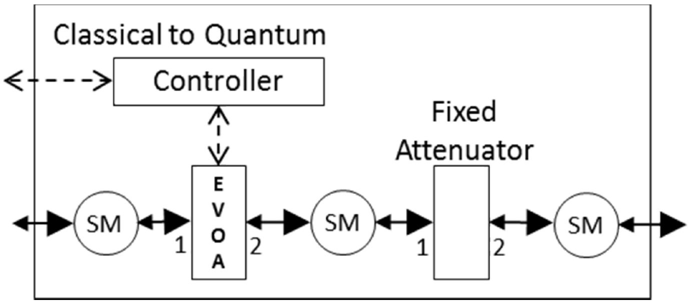


Figure 220. Classical To Quantum (CTQ) in the QKD system architecture.

The CTQ subsystem contains an electronically variable optical attenuator, a fixed optical attenuator and interconnecting SM optical fiber. We briefly discuss the behavior of each of the components contained within the module.

24.1.1 EVOA

The EVOA is an opto-electrical device containing a variable attenuator and support electronics to vary the output attenuation. The EVOA attenuates the power of optical signals by a variable amount. These devices usually have some form of blocking material such as an opaque slab or a window tilted in the path of the light. This blocking material is connected to an electric motor controlled by the higher system functions, allowing for a variable amount of light to exit the device (OZOptics, 2013; ThorLabs, 2013c). Optical signals arriving at one port propagate to the other port after a defined propagation delay with the attenuation based on the current controlled value.

X.1.2 Fixed Attenuator

The fixed attenuator is an optical device with two bidirectional optical ports that attenuates moving through the component. They are typically fabricated using either doped fibers or misaligned splices. The alternative build out-style attenuator is a small male-female adapter used to adjust the level of attenuation by coupling one or more FAs between fiber cables (ThorLabs, 2013a). Optical signals arriving at one port propagate to the other port after a defined propagation delay. The output of the fixed attenuator couples to the input of the isolator in the next subsystem using SM fiber.

X.1.3 Single-Mode Optical Fiber

SM fiber is an optical component used to interconnect optical devices. It has two bidirectional optical ports. Optical signals arriving at one port propagate to the other port after a defined propagation delay with its attenuation a function of the type and the length of the fiber. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass. It has a core which guides the light and an outer cladding that reflects the internal light back into the core, bouncing the light down the fiber. This cladding helps to reflect outside light to keep in from entering the core. This structure allows for low loss over long distances. The single-mode of the fiber comes from using a small core diameter (~10µm @ 1550nm) and small numerical aperture with the fundamental mode having a bell-shaped spatial distribution similar (Saleh & Teich, 1991; ThorLabs, 2013b). SM fiber couples devices within the module.

X.2 CTQ and Controller Behavior

The controller and individual components are sensitive to the temperature in the environment in which they operate. If the temperature exceeds defined thresholds, the components may become temporarily degraded or permanently damaged which changes their characteristics. If temporarily degraded, the devices may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with modeling the controller and module is to collect and understand the physical, behavioral, and performance characteristics of the atomic components. In this case, the individual components were constructed earlier and the controller was built as a message handler. The logic for the controller was based on the types of messages necessary for control of components inside the module.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

X.3 CTQ Compound Conceptual Model

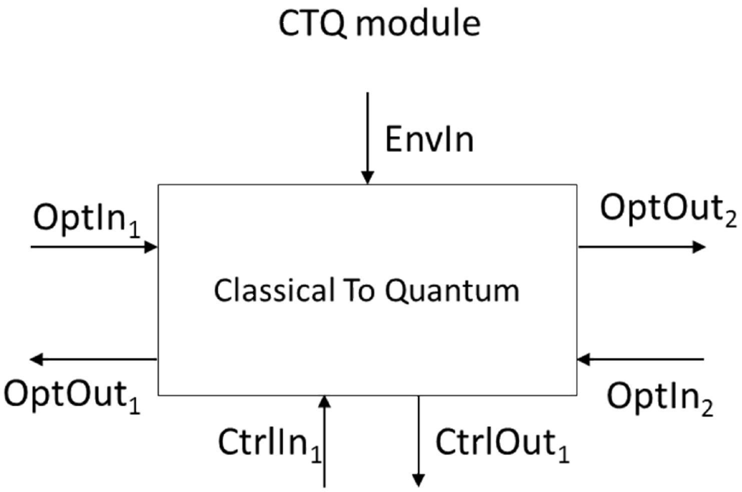


Figure 221. CTQ compound module conceptual model.

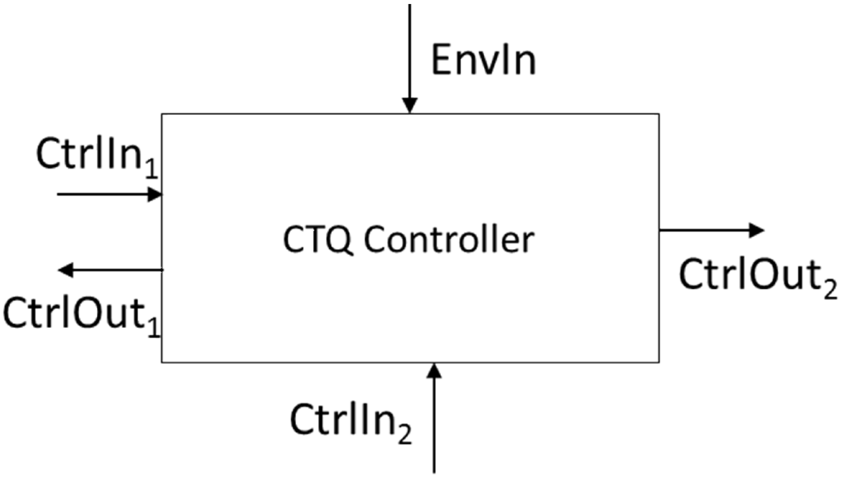


Figure 222. CTQ controller conceptual model

Table 92. *List of CTQ Controller messages.*

Input Messages	**From**	**Response**
CTQ_ENV	Quantum controller	Set the internal controller temperature
CTQ_RESET	Quantum controller	Resets the controller and clears the state variables
CTQ_STATUS_REQUEST	Quantum controller	Sends the controller status
CTQ_INCREASE_ATTEN	Quantum controller	Increases the attenuation
CTQ_DECREASE_ATTEN	Quantum controller	Decreases the attenuation
CTQ_SET_ATTEN	Quantum controller	Sets the attenuation

CTQ_GET_ATTEN	Quantum controller	Gets the current attenuation value
Output Messages	**To**	**Content**
CTQ_ACK	Quantum Controller	Response to a Reset message
CTQ_STATUS	Quantum Controller	Response to a Status Request message

The conceptual model for a CTQ consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_1$}, one environmental input port {EvnIn}, one control input port {CtrlIn$_1$} and one control output port {CtrlOut$_1$}. The environmental port allows external sources to communicate changes in the operational environment to the module. The electrical controller ports allow for control inputs to the controller and responses from the module to the higher system functions.

In comparison to the module layout used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism. The electrical control port is also decomposed in the model into an input port and an output port.

When an optical signal is sent to the input of the module, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn$_1$ in Fig. 2 will instantaneously generate a reflected emitting out of OptOut$_1$.

The module components must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters each of the components the module. In the case of optical overpowering, once overpowered a component will permanently change attenuation.

External environmental messages sent to the module are directed to individual components to convey the temperature of the operational environmental so the module can determine if it is degraded (a temporary condition) or damaged (a permanent condition). Changes to components based on the temperature determine the behavior of the module.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

X.4 English-Language Rules for the Controller

In this section, English language rules are developed to express the desired behavior of the controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When a control signal arrives:

- Determine the arrival port of the signal.
- Evaluate the content of the message
- Generate a response message to the incoming signal (if necessary).
- Generate a forwarded message to the appropriate device (if necessary).
- Output the response or forwarded message out the appropriate port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

X.5 DEVS Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the module controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs.

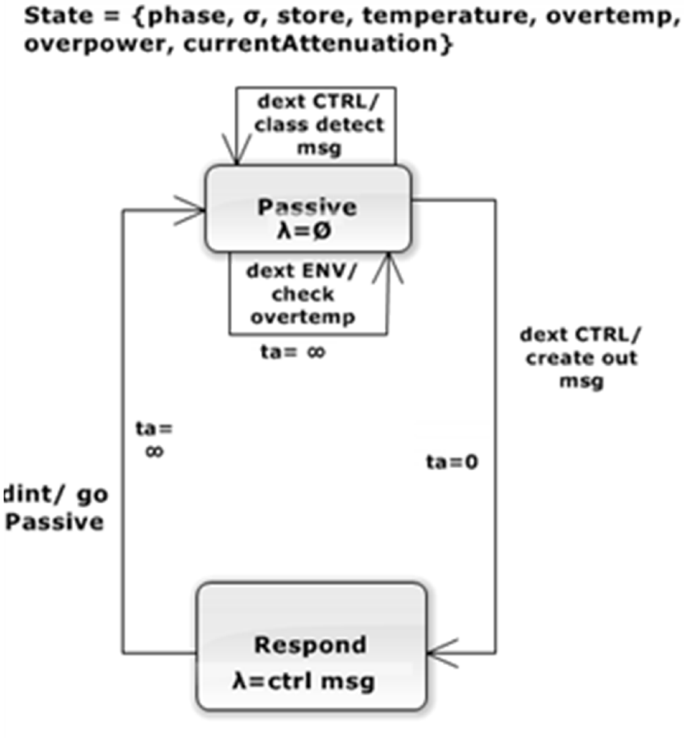

State = {phase, σ, store, temperature, overtemp, overpower, currentAttenuation}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 223. CTQ Controller DEVS phase transition diagram

X.6 CTQ Controller Event-Trace Diagram

This section shows various examples of messages entering the controller. The tables list the states the component proceeds through as the events are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the component, the over temperature flag variable and the over power flag variable. The queue column shows the contents of the queue at that state, the contents of the store state variable and any notes. Note in contrast to most other components, the controller is very simple and only responds to incoming messages; it does not generate any messages on its own. There are two types of inputs: control messages and environmental messages.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Current Attenuation: current EVOA attenuation setting
- Notes: any notes for that state

X.6.1 CASE I: Initial Passive with Single Control Packet Arriving at Time 0

Table 93. *Case I state list.*

time	state	entry/ exit	phase	sigma	store (*xi*)	temp	over temp	over power	current attenuation	Notes: assume tp=0
	1-packet	no env	no ext	1 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	s1	entry	respond	0	null	c	n	n	null	
0	s1	exit	respond	inf	null	c	n	n	null	
0	s2	entry	passive	inf	null	c	n	n	null	

X.6.2 CASE II: Initial Passive with Single Environmental Packet Arriving at Time 0

Table 94. *Case II state list.*

										Notes:
		entry/			**store**		**over**	**over**	**current**	**assume**
time	**state**	**exit**	**phase**	**sigma**	**(*xi*)**	**temp**	**temp**	**power**	**attenuation**	**tp=5**
	1-packet	1 env	no ext	0 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	S1	entry	passive	inf	null	c	n	n	null	

X.7 CTQ Controller Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

Definitions:

State = {phase, time advance, "store", temperature, "overtemp", "overpower", "currentAttenuation"}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level
"overpower" = flag variable set when device meets or exceeds damaged optical power level
"interruptRespond" = flag variable set when device is interrupted by an external event
"currentAttenuation" = variable to store the current attenuation value of the EVOA

For the controller we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;

$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;

S = set of sequential states;

$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;

$\delta_{int} = S \rightarrow S$ is the internal state transition function;

$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;

$\lambda = S \rightarrow Y^b$ is the output function;

$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;

$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

$\boldsymbol{DEVS_{CTQcontroller} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$

where

t_p = transmission time inside the component
temperature = current temperature of the component
phase = control state that keeps track of the internal phase of the component
phase = {“passive”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
currentAttenuation = variable that holds the current attenuation
interruptRespond = flag variable set when Respond phase is interrupted by an external event
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
ctrlOutput = variable that stores the output control message response
output.port = variable that holds the output optical packet port
store = variable that holds values of the current input values
timeLeftRespond = time left in Respond phase for the current event
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
ctrlMsg() = method that generates a response message to received control messages
messagebag_first() = method that returns the first element of the message bag
remove_event_m() = method that remove the current (x_i, time delay$_i$) from *messagebag*

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$” “EnvIn”} with
X_M = {(“CtrlIn$_1$”, V_{ctrl}), (“CtrlIn$_2$”, V_{ctrl}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”} with
Y_M = {(“CtrlOut$_1$”, $Y_{CtrlOut1}$), (“CtrlOut$_2$”, $Y_{CtrlOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, currentAttenuation* } = {{“passive”, “respond”} x R_0^+ x V x R x {“Y”, “N”} x {“Y”,”N”} x V }

External Transition Function:

```
$\delta_{ext}(phase, \sigma, store, temperature, overtemp, overpower, currentAttenuation ,e, ((p_i,v_i),…. (p_n,v_n)))$
                                                                                =
(“respond”, 0, $store, temperature, overtemp, overpower, currentAttenuation$)
   if $phase$ = “passive” and $p$ = “CtrlIn$_1$”
     $ctrlOutput$ = ctrlMsg($store$)
     if $ctrlMsg.status$ = “init” or “get status” or “get attenuation”
        $outputPort$ = “CtrlOut$_1$”
     if $ctrlMsg.status$ = “increase” or “decrease” or set”
        $outputPort$ = “CtrlOut$_2$”

(“passive”, 0, $store, temperature, overtemp, overpower, currentAttenuation$)
   if $phase$ = “passive” and $p$ = “CtrlIn$_2$”
      $currentAttenuation = messagebag.attenuation$

(“passive”, ∞, $store, temperature, overtemp, overpower, currentAttenuation$)
    if $phase$ = “passive” and $p$ = “EnvIn”
     $temperature = messagebag.temperature$
     if $temperature > damage.temp$
        $overtemp$ = “Y”

($phase, \sigma - e, store, temperature, overtemp, overpower, currentAttenuation$)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}(phase, \sigma, store, temperature, overtemp, overpower, currentAttenuation)$ =
  (“passive”, ∞, $store, temperature, overtemp, overpower, currentAttenuation$)
    if $phase$ = “respond”
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:

```
$\lambda(phase, \sigma, store, temperature, overtemp, overpower, currentAttenuation)$ =
  ($outputPort, ctrlOutput$)
      if phase = “respond”

   ∅ (null output)
     otherwise;
```

Time advance Function:

$ta(phase, \sigma, store, temperature, overtemp, overpower, currentAttenuation) = \sigma$;

X.8 CTQ Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

For the CTQ compound module we define:

Parallel-DEVS *compound* $N= (X, Y, D, \{M_d \mid \mathrm{d} \in D\}, EIC, EOC, IC)$

Where:

$X = \{(p,v) \mid \mathrm{p} \in IPorts, v \in X_p\}$ is the set of input ports and values;
$Y = \{(p,v) \mid \mathrm{p} \in OPorts, v \in Y_p\}$ is the set of output ports and values;
D = set of component names;
$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ is a DEVS atomic model;
$X_d = \{(p,v) \mid \mathrm{p} \in IPorts, v \in X_p\}$;
$Y_d = \{(p,v) \mid \mathrm{p} \in OPorts, v \in Y_p\}$;
$EIC \subseteq \{((N, ip_N),(d,ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in Iports_d\}$;
$EOC \subseteq \{((d,op_d),(N,op_N)) \mid op_N \in OPorts, d \in D, op_d \in Oports_d\}$;
$IC \subseteq \{((a,op_a),(b,ip_b)) \mid a,b \in D, op_a \in Oports_a, ip_b \in Iports_b\}$;
$((d,op_d),(e,ip_d)) \in IC$ implies $d \neq e$ (no feedback loops);
M = an atomic instance of P-DEVS.
N = a compound instance of P-DEVS.

$DEVS_{CTQ} = (X, Y, D, \{M_d \mid \mathrm{d} \in D\}, EIC, EOC, IC)$

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$”, “OptIn$_1$”, “OptIn$_2$”, “EnvIn”}
X = {(“CtrlIn$_1$”, v), (“CtrlIn$_2$”, v), (“OptIn$_1$”, v), (“OptIn$_2$”, v), (“EnvIn”, v) |$v \in V$}

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”, “OptOut$_1$”, “OptOut$_2$”}
Y = {(“CtrlOut$_1$”, v), (“CtrlOut$_2$”, v), (“OptOut$_1$”, v), (“OptOut$_2$”, v)|$v \in V$}

D = {controller, evoa, fixedattenuator, SMfiber$_1$, SMfiber$_2$, SMfiber$_3$}
$M_d = M_{controller}, M_{evoa}, M_{fixedattenuator}, Ms_{Mfiber1}, M_{SMfiber2}, Ms_{Mfiber3}$

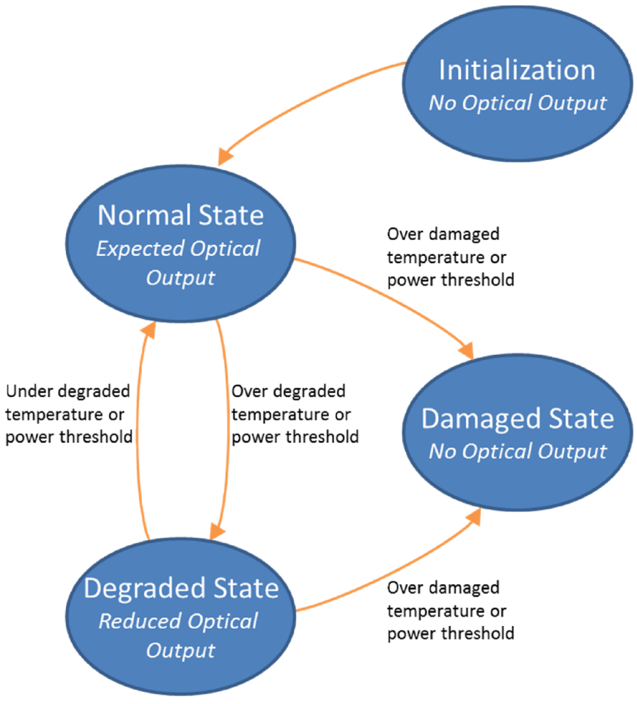
EIC = {((N, “CtrlIn$_1$”),(controller, “CtrlIn$_1$”)), ((N, “EnvIn”),(controller, “EnvIn”)), ((N, “EnvIn”),(evoa, “EnvIn”)), ((N, “EnvIn”),(fixedattenuator, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_1$, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_2$, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_3$, “EnvIn”)), ((N, “OptIn$_1$”),(SMfiber$_1$, “OptIn$_1$”)), ((N, “OptIn$_2$”),(SMfiber$_3$, “OptIn$_2$”))}

EOC = {((SMfiber$_1$, “OptOut$_1$”),(N, “OptOut$_1$”)), ((controller, “CtrlOut$_1$”),(N, “CtrlOut$_1$”)), ((SMfiber$_3$, “OptOut$_2$”),(N, “OptOut$_2$”))}

IC = {((controller, “CtrlOut$_2$”),(evoa, “CtrlIn$_1$”)), ((evoa, “CtrlOut$_1$”),(controller, “CtrlIn$_2$”)), ((SMfiber$_1$, “OptOut$_2$”),(evoa, “OptIn$_1$”)), ((evoa, “OptOut$_1$”),(SMfiber$_1$, “OptIn$_2$”)), ((evoa, “OptOut$_2$”),(SMfiber$_2$, “OptIn$_1$”)), ((SMfiber$_2$, “OptOut$_1$”),(evoa, “OptIn$_2$”)), ((SMfiber$_2$, “OptOut$_2$”),(fixedattenuator, “OptIn$_1$”)), ((fixedattenuator, “OptOut$_1$”),(SMfiber$_2$, “OptIn$_2$”)), ((fixedattenuator, “OptOut$_2$”),(SMfiber$_3$, “OptIn$_1$”)), ((SMfiber$_3$, “OptOut$_1$”),(fixedattenuator, “OptIn$_2$”))}

X.9 CTQ Controller Use Cases

X.9.1 Respond to a Quantum Controller Message


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 224. Component states.

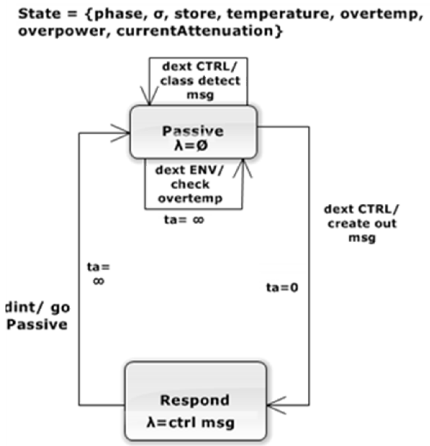

State = {phase, σ, store, temperature, overtemp, overpower, currentAttenuation}
dext CTRL/ class detect msg
Passive λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond λ=ctrl msg

Figure 225. Controller phase transition diagram

X.9.2 Respond to a Reset Message

Incoming reset message arrives at the module from the quantum controller. Pass the message to the module controller. Controller clears any stored variable values and prepares an acknowledgement message. Response message is sent out the appropriate port.

- Identified Alternative Uses Cases
  - React to an environmental message
  - React to a status request message
  - React to an increase attenuation message
  - React to a decrease attenuation message
  - React to a set attenuation message
  - React to a get attenuation message
- Assumptions
  - Incoming electrical signals are not affected by component state

X.9.3 Respond to Reset Message End Goals

- Message properly received
- Controller enters Respond phase and sets storage values to zero.
- Controller forwards Reset Message to proper component(s) as necessary
- Acknowledgement message created and sent out the appropriate port
- Controller ends in Passive phase

X.9.4 Respond to an Environmental Packet

Environmental packet arrives at the controller. Check to see if environmental packet temperature sets the controller to degraded or damaged state. Check to see if temperature level returns controller from degraded state to normal state. Records change in condition, if applicable. Change controller function if in degraded or damaged state, if necessary.

- Assumptions
    - None

X.9.5 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

X.9.6 Respond to a Status Request Message

Status Request message arrives at the module from the quantum controller. Module controller prepares response message. Response message is sent out the appropriate port.

- Assumptions
    - Controller has completed initialization sequence at least once

X.9.7 Respond to Status Request End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

X.9.8 Respond to an Increase Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
    - Controller has completed initialization sequence at least once

X.9.9 Respond to Increase Attenuation Message End Goals

- Increase Attenuation message received properly
- Message recognized and passed to proper component

X.9.10 Respond to a Decrease Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

X.9.11 Respond to Decrease Attenuation Message End Goals

- Decrease Attenuation message received properly
- Message recognized and passed to proper component

X.9.12 Respond to a Set Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

X.9.13 Respond to Set Attenuation Message End Goals

- Set Attenuation message received properly
- Message recognized and passed to proper component

X.9.14 Respond to a Get Attenuation Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

X.9.15 Respond to Get Attenuation Message End Goals

- Get Attenuation message received properly
- Message recognized and passed to proper component

X.10 CTQ Module Use Cases

X.10.1 Respond to an Optical Packet

Optical packet arrives at the module. Pass the optical packet to the proper internal component.

- Assumptions
    - Reflections are not affected by module or component state

X.10.2 Respond to Optical Packet End Goals

- Optical packet sent to proper internal component

X.10.3 Respond to an Environmental Message

Environmental packet arrives at the module. Environmental message is passed to the module controller and each component in the module.

- Assumptions
    - Incoming electrical signals are not affected by component state

X.10.4 Respond to Environmental Message End Goals

- Environmental packet received properly and forwarded to each component

X.10.5 Respond to a Control Message

Control message arrives at the module. Control message is passed to the module controller.

- Assumptions
    - Incoming electrical signals are not affected by component state

X.10.6 Respond to Environmental Message End Goals

- Control message received properly and forwarded to the module controller

X.11 CTQ Test Cases

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The type of control message passed to each coupled submodule depended on the internal components.

- CTQ submodule – control message changes attenuation of EVOA

These test cases led to iterations of testing and correction. Optical messages were tracked through the internal components and out the submodule output. Environmental messages were checked to ensure they replicated to each internal component. All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

	total tests	optical ports	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

X.12 References

OZOptics. (2013). Electrically-controlled variable fiber-optic attenuator. Retrieved, 2013, Retrieved from http://www.ozoptics.com/ALLNEW_PDF/DTS0010.pdf

Saleh, B. E. A., & Teich, M. C. (1991). Guided waves. *Fundamentals of photonics* (2nd ed., pp. 340-342). New York: John Wiley & Sons, Inc.

ThorLabs. (2013a). Fixed fiber optical attenuators. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1385

ThorLabs. (2013b). Single-mode fiber. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=949

ThorLabs. (2013c). Variable fiber optical attenuators, single mode. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6161

Appendix Y - Optical Security Layer (OSL)

Y.1 Device Description:

The optical security layer (OSL) works to detect and limit the amount of outside light entering the QKD system. The bandpass filter narrows the incoming light frequencies and the circulator routes the incoming light to the classical detector. The detector acts as the 'alarm' by creating an electrical signal to the quantum controller. The circulator allows very little light to pass from port one to three and any that does is heavily attenuated by the isolator. The OSL subsystem contains the components shown in Fig. 1.

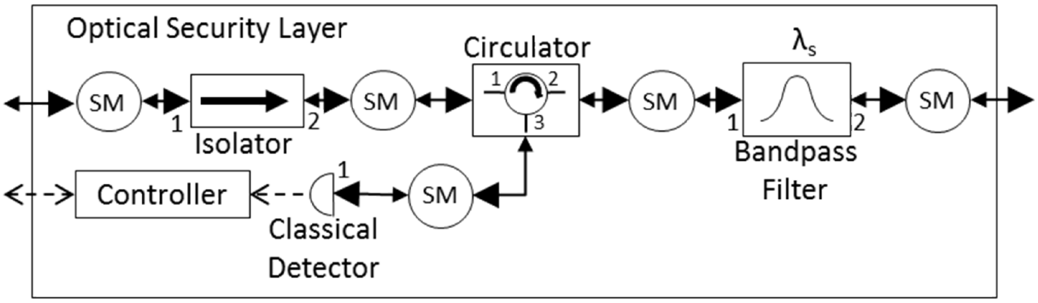


Figure 226. Optical Security Layer (OSL) in the QKD system architecture.

The OSL subsystem contains a controller, an isolator, a circulator, a bandpass filter, a classical detector, electrical interfaces, and interconnecting SM optical fiber. We briefly discuss the behavior of each of the components contained within the OSL.

Y.1.1 OSL Controller

The controller is an electrical device containing digital and analog circuits responsible for monitoring the classical detector. It has a bidirectional electrical interface to the quantum module controller and an electrical input from the classical detector. It receives commands from the quantum model controller and stores information from the classical detector.

Y.1.2 Isolator

The isolator is an optical device with two bidirectional optical ports that passes light in the forward direction while significantly attenuating light moving in the opposite direction (ThorLabs, 2013b). Optical signals arriving at one port propagate to the other port after a defined propagation delay with the attenuation based on the propagation direction. The isolator assures that virtually no light (e.g., reflections or light from external sources) enters the laser. The output of the isolator is coupled to the input of the circulator via SM fiber

Y.1.3 Circulator

The circulator is an optical device with three bidirectional optical ports that allows light to pass through in one direction. Light entering port one exits port two with minimal attenuation but is highly attenuated leaving port three. Light entering port two exits port three with minimal attenuation and is highly attenuated leaving port one. A "full" circulator allows light to enter port three and pass on to port one with minimal attenuation but heavily attenuating port two. A "quasi" circulator highly attenuates any light entering port three at ports one and two (ThorLabs, 2013d). In the OSL, the circulator directs light from the laser in port 1 and out port 2 out to the bandpass filter. Light entering port 2 directs to port 3 and on to the classical detector.

Y.1.4 Bandpass Filter

The bandpass filter is an optical device with two bidirectional optical ports that passes the optical energy in a narrow band around the signal wavelength, λ_S, but strongly attenuates other wavelengths (ThorLabs, 2013a). This ensures that only the appropriate signal wavelength leaves the subsystem while preventing other sources of light from entering the laser. Optical signals arriving at one port propagate to the other port after a defined propagation delay and are attenuated based on the wavelength of the signal.

Y.1.5 Classical Detector

The classical detector is an opto-electrical device containing an optical photodiode and support electronics to generate an electrical signal proportional to the power contained in the optical pulse (ThorLabs, 2013c). This signal connects to the controller which stores this information and checks to see if it falls below a predefined threshold. If so, the controller notifies the quantum module controller of incoming light and functions as an alert device.

Y.1.6 Single-Mode Optical Fiber

SM fiber is an optical component used to interconnect optical devices. It has two bidirectional optical ports. Optical signals arriving at one port propagate to the other port after a defined propagation delay with its attenuation a function of the type and the length of the fiber. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass. It has a core which guides the light and an outer cladding that reflects the internal light back into the core, bouncing the light down the fiber. This cladding helps to reflect outside light to keep in from entering the core. This structure allows for low loss over long distances. The single-mode of the fiber comes from using a small core diameter (~10µm @ 1550nm) and small numerical aperture with the fundamental mode having a bell-shaped spatial distribution similar (Saleh & Teich, 1991; ThorLabs, 2013e). SM fiber couples devices within the module.

Y.2 OSL and Controller Behavior

The controller and individual components are sensitive to the temperature in the environment in which they operate. If the temperature exceeds defined thresholds, the components may become temporarily degraded or permanently damaged which changes their characteristics. If temporarily degraded, the devices may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with modeling the controller and module is to collect and understand the physical, behavioral, and performance characteristics of the atomic components. In this case, the individual components were constructed earlier and the controller was built as a message handler. The logic for the controller was based on the types of messages necessary for control of components inside the module.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

Y.3 OSL Compound Conceptual Model

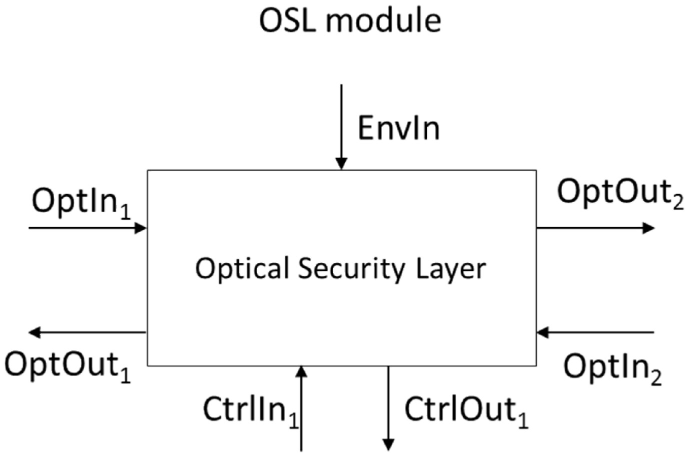


Figure 227. OSL compound module conceptual model.

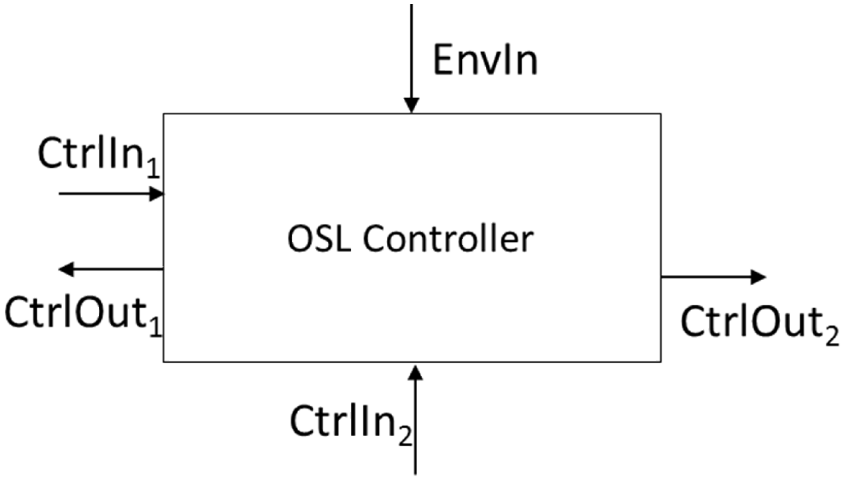


Figure 228. OSL controller conceptual model

Table 95. *List of OSL Controller messages*.

Input Messages	**From**	**Response**
OSL_ENV	Quantum controller	Set the internal CPG controller temperature
OSL_RESET	Quantum controller	Resets the CPG controller and clears the state variables
OSL_STATUS_REQUEST	Quantum controller	Sends the CPG controller status and stored magnitude value
CD_DETECTION	Classical Detector	Store the magnitude from the message
Output Messages	**To**	**Content**
OSL_ACK	Quantum Controller	Response to a Reset message
OSL_STATUS	Quantum Controller	Response to a Status Request message

The conceptual model for the OSL consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, one environmental input port {EvnIn}, one control input port {CtrlIn$_1$} and one control output port {CtrlOut$_1$}. The environmental port allows external sources to communicate changes in the operational environment to the module. The electrical controller ports allow for control inputs to the controller and responses from the module to the higher system functions.

In comparison to the module layout used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism. The electrical control port is also decomposed in the model into an input port and an output port.

When an optical signal is sent to the input of the module, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn_1 in Fig. 2 will instantaneously generate a reflected emitting out of OptOut_1.

The module components must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters each of the components the module. In the case of optical overpowering, once overpowered a component will permanently change attenuation. External environmental messages sent to the module are directed to individual components to convey the temperature of the operational environmental so the module can determine if it is degraded (a temporary condition) or damaged (a permanent condition). Changes to components based on the temperature determine the behavior of the module.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

Y.4 English-Language Rules for the Controller

In this section, English language rules are developed to express the desired behavior of the controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When a control signal arrives:

- Determine the arrival port of the signal.
- Evaluate the content of the message
- Generate a response message to the incoming signal (if necessary).
- Generate a forwarded message to the appropriate device (if necessary).
- Output the response or forwarded message out the appropriate port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

Y.5 DEVS Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the module controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs.

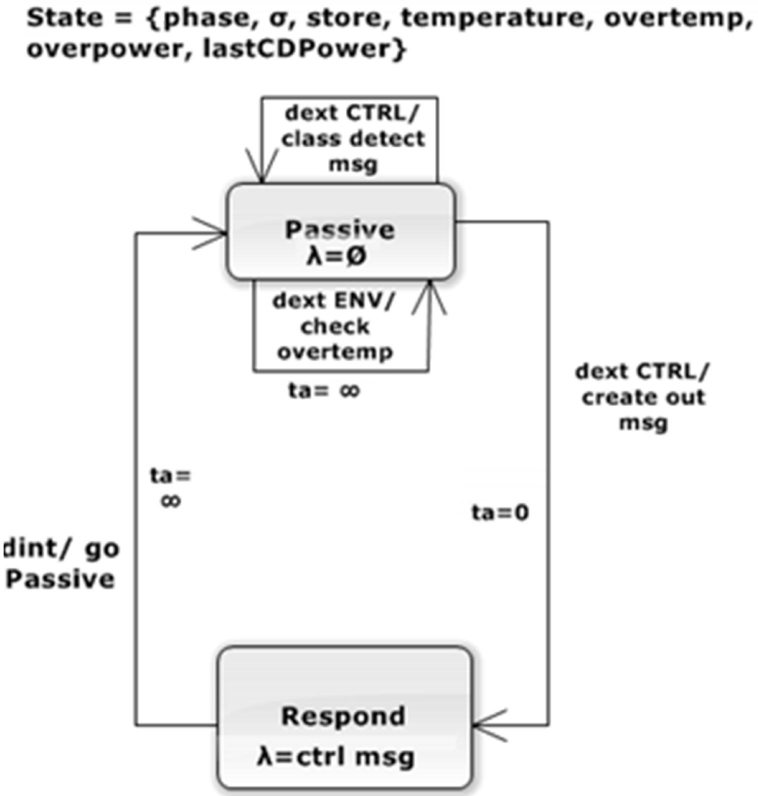

State = {phase, σ, store, temperature, overtemp, overpower, lastCDPower}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 229. OSL Controller DEVS phase transition diagram

Y.6 OSL Controller Event-Trace Diagram

This section shows various examples of messages entering the controller. The tables list the states the component proceeds through as the events are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the component, the over temperature flag variable and the over power flag variable. The queue column shows the contents of the queue at that state, the contents of the store state variable and any notes. Note in contrast to most other components, the controller is very simple and only responds to incoming messages; it does not generate any messages on its own. There are two types of inputs: control messages and environmental messages.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- LastCDPower: shows the value of the last detection message
- Notes: any notes for that state

Y.6.1 CASE I: Initial Passive with Single Control Packet Arriving at Time 0

Table 96. *Case I state list.*

		entry/			**store**		**over**	**over**		**Notes:**
time	**state**	**exit**	**phase**	**sigma**	**(*xi*)**	**temp**	**temp**	**power**	**lastCD power**	**assume tp=0**
	1-packet	no env	no ext	1 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	s1	entry	respond	0	null	c	n	n	null	
0	s1	exit	respond	inf	null	c	n	n	null	
0	s2	entry	passive	inf	null	c	n	n	null	

Y.6.2 CASE II: Initial Passive with Single Environmental Packet Arriving at Time 0

Table 97. *Case II state list.*

		entry/			**store**		**over**	**over**		**Notes:**
time	**state**	**exit**	**phase**	**sigma**	**(*xi*)**	**temp**	**temp**	**power**	**lastCD power**	**assume tp=0**
	1-packet	1 env	no ext	0 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	S1	entry	passive	inf	null	c	n	n	null	

Y.7 OSL Controller Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

Definitions:

State = {phase, time advance, "store", temperature, "overtemp", "overpower", "lastCDPower"}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
"store" = state variable that stores the current input values
"overtemp" = flag variable set when device meets or exceeds damaged temperature level
"overpower" = flag variable set when device meets or exceeds damaged optical power level
"interruptRespond" = flag variable set when device is interrupted by an external event
"lastCDPower" = variable to store the power value of the last classical detector message

For the controller we define:

Parallel-DEVS *atomic M*= (X_M, Y_M, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta)

Where:

X_M = $\{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
Y_M = $\{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

***DEVS**$_{OSLcontroller}$* **=** ($\boldsymbol{X_M}$, $\boldsymbol{Y_M}$, $\boldsymbol{S}$, $\boldsymbol{\delta_{ext}}$, $\boldsymbol{\delta_{int}}$, $\boldsymbol{\delta_{con}}$, $\boldsymbol{\lambda}$, $\boldsymbol{ta}$)

where

t_p = transmission time inside the component
temperature = current temperature of the component
phase = control state that keeps track of the internal phase of the component
phase = {"passive", "respond"}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
lastCDPower= variable that holds the magnitude of the last detection message
interruptRespond = flag variable set when Respond phase is interrupted by an external event

messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
ctrlOutput = variable that stores the output control message response
output.port = variable that holds the output optical packet port
store = variable that holds values of the current input values
timeLeftRespond = time left in Respond phase for the current event
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
ctrlMsg() = method that generates a response message to received control messages
messagebag_first() = method that returns the first element of the message bag
remove_event_m() = method that remove the current (x_i, time delay_i) from *messagebag*

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"CtrlIn_1", "CtrlIn_2" "EnvIn"} with
X_M = {("CtrlIn_1", V_{ctrl}), ("CtrlIn_2", V_{ctrl}), ("EnvIn", V_{env})} is the set of input ports and values.

OutPorts = {"CtrlOut_1"} with
Y_M = {("CtrlOut_1", $Y_{CtrlOut1}$) is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, lastCDPower*} = {{"passive", "respond"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x *V*}

External Transition Function:

```
δ_ext(phase, σ, store, temperature, overtemp, overpower, lastCDPower ,e, ((p_i,v_i),…. (p_n,v_n))) =
("respond", 0, store, temperature, overtemp, overpower, lastCDPower)
  if phase = "passive" and p = "CtrlIn_1"
    ctrlOutput = ctrlMsg(store)
    if ctrlMsg.status = "init" or "get status"
      outputPort = "CtrlOut_1"

("passive", 0, store, temperature, overtemp, overpower, lastCDPower)
  if phase = "passive" and p = "CtrlIn_2"
    lastCDPower = messagebag.magnitude

("passive", ∞, store, temperature, overtemp, overpower, lastCDPower)
  if phase = "passive" and p = "EnvIn"
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = "Y"
```

(*phase*, $\sigma - e$, *store, temperature*, *overtemp*, *overpower*, *lastCDPower*)
otherwise;

Internal Transition Function:

δ_{int}(*phase*, σ, *store, temperature*, *overtemp*, *overpower*, *lastCDPower*) =
(“passive”, ∞, *store, temperature*, *overtemp*, *overpower*, *lastCDPower*)
if *phase* = “respond”

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:
λ(*phase*, σ, *store, temperature*, *overtemp*, *overpower*, *lastCDPower*) =
(*outputPort*, *ctrlOutput*)
if phase = “respond”

$\emptyset$ (null output)
otherwise;

Time advance Function:
ta(*phase*, σ, *store, temperature*, *overtemp*, *overpower*, *lastCDPower*) = σ;

Y.8 OSL Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

For the OSL compound module we define:

Parallel-DEVS *compound N*= (*X, Y, D,* $\{M_d \mid \mathrm{d} \in D\}$, *EIC*, *EOC*, *IC*)

Where:

$X = \{(p,v) \mid \mathrm{p} \in IPorts, v \in X_p\}$ is the set of input ports and values;
$Y = \{(p,v) \mid \mathrm{p} \in OPorts, v \in Y_p\}$ is the set of output ports and values;
D = set of component names;

$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ is a DEVS atomic model;
$X_d = \{(p,v) \mid \mathrm{p} \in IPorts, v \in X_p\}$;
$Y_d = \{(p,v) \mid \mathrm{p} \in OPorts, v \in Y_p\}$;
$EIC \subseteq \{((N, ip_N),(d,ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in Iports_d\}$;
$EOC \subseteq \{((d,op_d),(N,op_N)) \mid op_N \in OPorts, d \in D, op_d \in Oports_d\}$;
$IC \subseteq \{((a,op_a),(b,ip_b)) \mid a,b \in D, op_a \in Oports_a, ip_b \in Iports_b\}$;
$((d,op_d),(e,ip_d)) \in IC$ implies $d \neq e$ (no feedback loops);
M = an atomic instance of P-DEVS.
N = a compound instance of P-DEVS.

$$\boldsymbol{DEVS_{OSL} = (X, Y, D, \{M_d \mid \mathrm{d} \in D\}, EIC, EOC, IC)}$$

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$”, “OptIn$_1$”, “OptIn$_2$”, “OptIn$_3$”, “EnvIn”}
X = {(“CtrlIn$_1$”, v), (“CtrlIn$_2$”, v), (“OptIn$_1$”, v), (“OptIn$_2$”, v), (“OptIn$_3$”, v), (“EnvIn”, v) |$v \in V$}

OutPorts = {“CtrlOut$_1$”, “OptOut$_1$”, “OptOut$_2$”, “OptOut$_3$”}
Y = {(“CtrlOut$_1$”, v), (“OptOut$_1$”, v), (“OptOut$_2$”, v), (“OptOut$_3$”, v)|$v \in V$}

D = {controller, isolator, circulator, bandpass, classicaldetector, SMfiber$_1$, SMfiber$_2$, SMfiber$_3$, SMfiber$_4$, SMfiber$_5$}
$M_d = M_{controller}, M_{isolator}, M_{circulator}, M_{bandpass}, M_{classicaldetector}, Ms_{Mfiber1}, M_{SMfiber2}, Ms_{Mfiber3}, Ms_{Mfiber4}, Ms_{Mfiber5}$

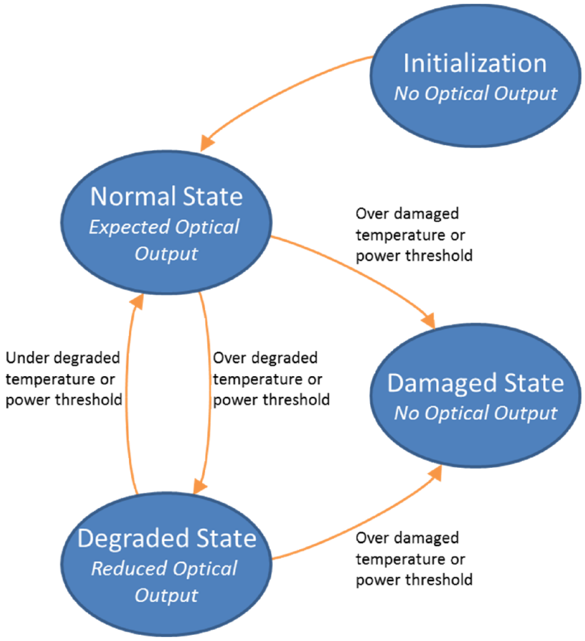
EIC = {((N, “CtrlIn$_1$”),(controller, “CtrlIn$_1$”)), ((N, “EnvIn”),(controller, “EnvIn”)), ((N, “EnvIn”),(isolator, “EnvIn”)), ((N, “EnvIn”),(circulator, “EnvIn”)), ((N, “EnvIn”),(bandpass, “EnvIn”)), ((N, “EnvIn”),(classicaldetector, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_1$, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_2$, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_3$, “EnvIn”)), (SMfiber$_4$, “EnvIn”)), (SMfiber$_5$, “EnvIn”)), ((N, “OptIn$_1$”),(SMfiber$_1$, “OptIn$_1$”)), ((N, “OptIn$_2$”),(SMfiber$_4$, “OptIn$_2$”))}

EOC = {((SMfiber$_1$, “OptOut$_1$”),(N, “OptOut$_1$”)), ((controller, “CtrlOut$_1$”),(N, “CtrlOut$_1$”)), ((SMfiber$_4$, “OptOut$_2$”),(N, “OptOut$_2$”))}

IC = {((classicaldetector, “CtrlOut$_1$”),(controller, “CtrlIn$_2$”)),
((SMfiber$_1$, “OptOut$_2$”),(isolator, “OptIn$_1$”)), ((isolator, “OptOut$_1$”),(SMfiber$_1$, “OptIn$_2$”)),
((isolator, “OptOut$_2$”),(SMfiber$_2$, “OptIn$_1$”)), ((SMfiber$_2$, “OptOut$_1$”),(isolator, “OptIn$_2$”)),
((SMfiber$_2$, “OptOut$_2$”),(circulator, “OptIn$_1$”)), ((circulator, “OptOut$_1$”),(SMfiber$_2$, “OptIn$_2$”)),
((circulator, “OptOut$_2$”),(SMfiber$_3$, “OptIn$_1$”)), ((SMfiber$_3$, “OptOut$_1$”),(circulator, “OptIn$_2$”))
((SMfiber$_3$, “OptOut$_2$”),(bandpass, “OptIn$_1$”)), ((bandpass, “OptOut$_1$”),(SMfiber$_3$, “OptIn$_2$”)),
((bandpass, “OptOut$_2$”),(SMfiber$_4$, “OptIn$_1$”)), ((SMfiber$_4$, “OptOut$_1$”),(bandpass, “OptIn$_2$”)),
((circulator, “OptOut$_3$”),(SMfiber$_5$, “OptIn$_2$”)), ((SMfiber$_5$, “OptOut$_2$”),(circulator, “OptIn$_3$”))
((SMfiber$_5$, “OptOut$_1$”),(classicaldetector, “OptIn$_1$”)), ((classicaldetector, “OptOut$_1$”),(SMfiber$_5$, “OptIn$_1$”))}

Y.9 OSL Controller Use Cases

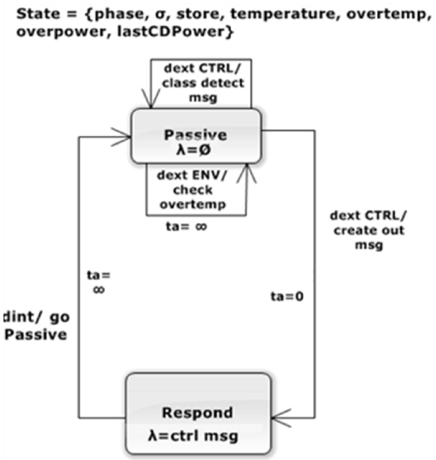
Y.9.1 Respond to a Quantum Controller Message



Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 230. Component states.



State = {phase, σ, store, temperature, overtemp, overpower, lastCDPower}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 231. Controller phase transition diagram

Y.9.2 Respond to a Reset Message

Incoming reset message arrives at the module from the quantum controller. Pass the message to the module controller. Controller clears any stored variable values and prepares an acknowledgement message. Response message is sent out the appropriate port.

- Identified Alternative Uses Cases
    - React to an environmental message
    - React to a status request message
    - React to a fire laser message
    - React to a classical detector pulse detection message

- Assumptions
    - Incoming electrical signals are not affected by component state

Y.9.3 Respond to Reset Message End Goals

- Message properly received
- Controller enters Respond phase and sets storage values to zero.
- Controller forwards Reset Message to proper component(s) as necessary
- Acknowledgement message created and sent out the appropriate port
- Controller ends in Passive phase

Y.9.4 Respond to an Environmental Packet

Environmental packet arrives at the controller. Check to see if environmental packet temperature sets the controller to degraded or damaged state. Check to see if temperature level returns controller from degraded state to normal state. Records change in condition, if applicable. Change controller function if in degraded or damaged state, if necessary.

- Assumptions
    - None

Y.9.5 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

Y.9.6 Respond to a Status Request Message

Status Request message arrives at the module from the quantum controller. Module controller prepares response message. Response message is sent out the appropriate port.

- Assumptions
  - Controller has completed initialization sequence at least once

Y.9.7 Respond to Status Request End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

Y.9.8 Respond to a Classical Detection Message

Incoming detection message arrives at the controller from the classical detector. Store the message contents.

- Assumptions
  - Controller has completed initialization sequence at least once

Y.9.9 Respond to Classical Detection Message End Goals

- CD message received properly
- CD message values stored properly

Y.10 OSL Module Use Cases

Y.10.1 Respond to an Optical Packet

Optical packet arrives at the module. Pass the optical packet to the proper internal component.

- Assumptions
  - Reflections are not affected by module or component state

Y.10.2 Respond to Optical Packet End Goals

- Optical packet sent to proper internal component

Y.10.3 Respond to an Environmental Message

Environmental packet arrives at the module. Environmental message is passed to the module controller and each component in the module.

- Assumptions
    - Incoming electrical signals are not affected by component state

Y.10.4 Respond to Environmental Message End Goals

- Environmental packet received properly and forwarded to each component

Y.10.5 Respond to a Control Message

Control message arrives at the module. Control message is passed to the module controller.

- Assumptions
    - Incoming electrical signals are not affected by component state

Y.10.6 Respond to Environmental Message End Goals

- Control message received properly and forwarded to the module controller

Y.11 OSL Test Cases

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The type of control message passed to each coupled submodule depended on the internal components.

- OSL submodule – no control message to change internal settings

These test cases led to iterations of testing and correction. Optical messages were tracked through the internal components and out the submodule output. Environmental messages were checked to ensure they replicated to each internal component. All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

	total tests	optical ports	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

Y.12 References

Saleh, B. E. A., & Teich, M. C. (1991). Guided waves. *Fundamentals of photonics* (2nd ed., pp. 340-342). New York: John Wiley & Sons, Inc.

ThorLabs. (2013a). Bandpass filter structure. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1000

ThorLabs. (2013b). Optical isolator tutorial. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6178

ThorLabs. (2013c). Photodiode tutorial. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=2822

ThorLabs. (2013d). Single mode fiber optic circulators. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=373

ThorLabs. (2013e). Single-mode fiber. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=949

Appendix Z - Timing Pulse Generator (TPG)

Z.1 Device Description:

The optical frames used by the QKD system need a way to synchronize between Alice and Bob. Even the best timing devices have error and other external timing (e.g. GPS) do not have the accuracy necessary for frame timing. Bob needs to know with a high-degree of accuracy when to open the gates windows for his single photon detectors. Alice provides this timing by injecting a bright pulse (i.e. classical level) of light (λ_t) into the quantum channel to start each frame. The wave division multiplexer multiplexes the TPG timing signal λ_t, and the CPG signal pulse λ_S. Once intermixed, the pulses output to the output monitor subsystem. The TPG contains the components shown in Fig. 1.

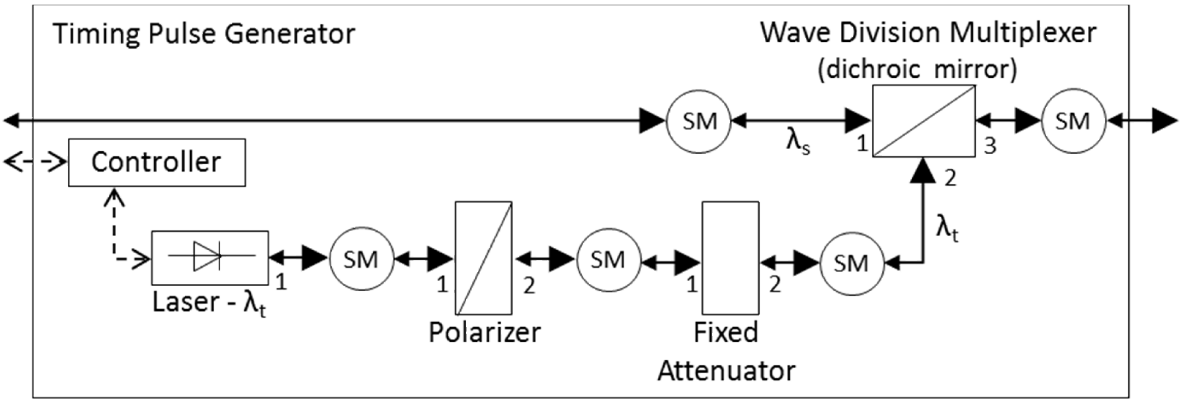


Figure 232. Timing Pulse Generator (TPG) in the QKD system architecture.

The TPG subsystem contains a controller, a laser, an optical polarizer, a fixed attenuator, wave division multiplexer, electrical interfaces, and interconnecting SM optical fiber. We briefly discuss the behavior of each of the components contained within the TPG.

Z.1.1 TPG Controller

The controller is an electrical device containing digital and analog circuits responsible for controlling the laser. It has a bidirectional electrical interface to the quantum module controller and an electrical output to the laser. It receives commands from the quantum model controller, sends fire commands to the laser, and monitors the health of the laser.

Z.1.2 Laser

The laser is an electro-optical device which contains an optical oscillator and emits coherent light (Saleh & Teich, 1991b). It has an electrical input to receive control messages and an optical output to emit generated pulses. Within the simulation, the laser creates optical pulses when it receives a “fire” command from the controller. The laser generates short-duration laser pulses (e.g., 1mW peak intensity with a 500ps duration) containing millions of photons (ThorLabs, 2013a). The output of the laser couples to the input of the polarizer via SM fiber.

Z.1.3 Polarizer

The polarizer is an optical device with two bidirectional optical ports allowing light of one polarization to pass while highly attenuating light orthogonal to the passed light (ThorLabs, 2013c). Optical signals arriving at one port propagate to the other port after a defined propagation delay and polarized depending on the polarizer orientation with respect to the connected fiber. The output of the polarizer is coupled to the input of the fixed attenuator via SM fiber.

Z.1.4 Fixed Attenuator

The fixed attenuator is an optical device with two bidirectional optical ports that attenuates moving through the component. They are typically fabricated using either doped fibers or misaligned splices. The alternative build out-style attenuator is a small male-female adapter used to adjust the level of attenuation by coupling one or more FAs between fiber cables (ThorLabs, 2013b). Optical signals arriving at one port propagate to the other port after a defined propagation delay. The output of the fixed attenuator couples to the input of the WDM using SM fiber.

Z.1.5 Wave Division Multiplexor (dichroic mirror)

The WDM is an optical device with three bidirectional optical ports used to combine (or split) different wavelengths of light into one stream. In the opposite direction through the WDM, combined optical signals are separated into different streams and sent out different ports (OZOptics, 2013). Here, the WDM combines the signal and timing pulse. Optical signals arriving at one port propagate to the other port after a defined propagation delay. The WDM output couples to input of the next subsystem via SM fiber.

Z.1.6 Single-Mode Optical Fiber

SM fiber is an optical component used to interconnect optical devices. It has two bidirectional optical ports. Optical signals arriving at one port propagate to the other port after a defined propagation delay with its attenuation a function of the type and the length of the fiber. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass. It has a core which guides the light and an outer cladding that reflects the internal light back into the core, bouncing the light down the fiber. This cladding helps to reflect outside light to keep in from entering the core. This structure allows for low loss over long distances. The single-mode of the fiber comes from using a small core diameter (~10µm @ 1550nm) and small numerical aperture with the fundamental mode having a bell-shaped spatial distribution similar (Saleh & Teich, 1991a; ThorLabs, 2013d). SM fiber couples devices within the module.

Z.2 TPG and Controller Behavior

The controller and individual components are sensitive to the temperature in the environment in which they operate. If the temperature exceeds defined thresholds, the components may become temporarily degraded or permanently damaged which changes their characteristics. If temporarily degraded, the devices may recover to normal operating behavior after the temperature returns to a “normal” operating temperature.

The first step involved with modeling the controller and module is to collect and understand the physical, behavioral, and performance characteristics of the atomic components. In this case, the individual components were constructed earlier and the controller was built as a message handler. The logic for the controller was based on the types of messages necessary for control of components inside the module.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

Z.3 TPG Compound Conceptual Model

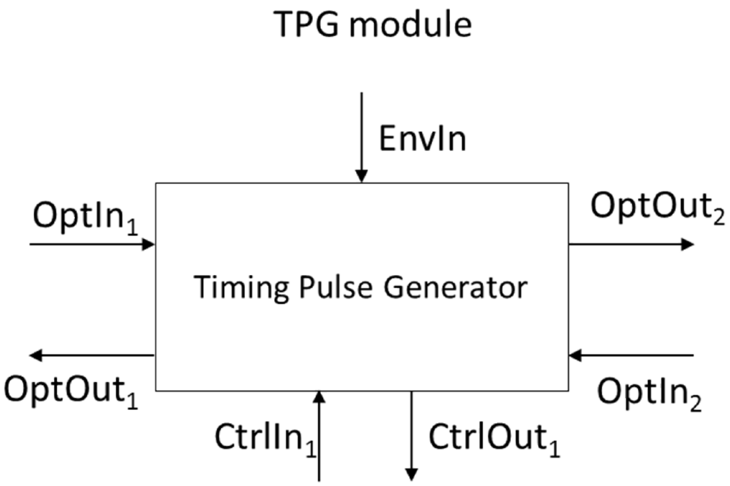


Figure 233. TPG compound module conceptual model.

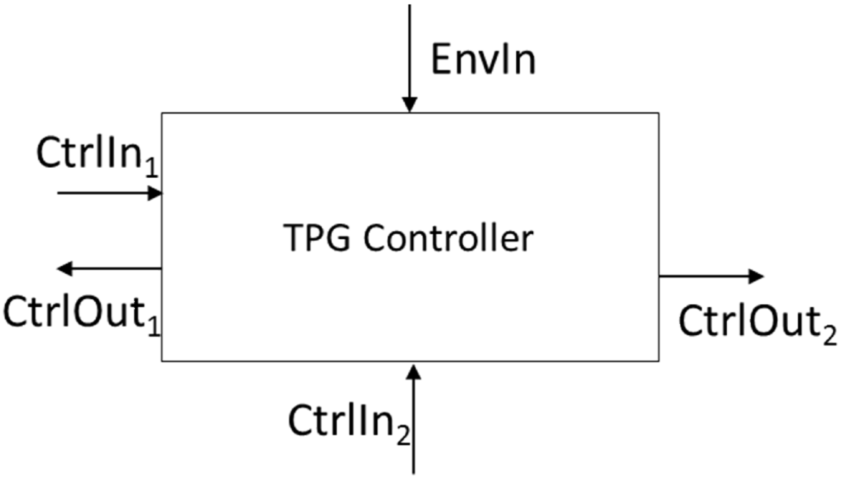


Figure 234. TPG controller conceptual model

Table 98. *List of TPG Controller messages*.

Input Messages	**From**	**Response**
TPG_ENV	Quantum controller	Set the internal CPG controller temperature
TPG_RESET	Quantum controller	Resets the CPG controller and clears the state variables
TPG_STATUS_REQUEST	Quantum controller	Sends the CPG controller status and stored magnitude value
TPG_FIRE_LASER	Quantum controller	Issues a single Fire Laser command to the laser

Output Messages	**To**	**Content**
TPG_ACK	Quantum Controller	Response to a Reset message
TPG_STATUS	Quantum Controller	Response to a Status Request message
TPG_LASER_FIRE	Laser	Command to fire the laser one time

The conceptual model for the TPG consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, one environmental input port {EvnIn}, one control input port {CtrlIn$_1$} and one control output port {CtrlOut$_1$}. The environmental port allows external sources to communicate changes in the operational environment to the module. The electrical controller ports allow for control inputs to the controller and responses from the module to the higher system functions.

In comparison to the module layout used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism. The electrical control port is also decomposed in the model into an input port and an output port.

When an optical signal is sent to the input of the module, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn$_1$ in Fig. 2 will instantaneously generate a reflected emitting out of OptOut$_1$.

The module components must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters each of the components the module. In the case of optical overpowering, once overpowered a component will permanently change attenuation. External environmental messages sent to the module are directed to individual components to convey the temperature of the operational environmental so the module can determine if it is degraded (a temporary condition) or damaged (a permanent condition). Changes to components based on the temperature determine the behavior of the module.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

Z.4 English-Language Rules for the Controller

In this section, English language rules are developed to express the desired behavior of the controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.
- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When a control signal arrives:

- Determine the arrival port of the signal.
- Evaluate the content of the message
- Generate a response message to the incoming signal (if necessary).
- Generate a forwarded message to the appropriate device (if necessary).
- Output the response or forwarded message out the appropriate port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

Z.5 DEVS Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the module controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs.

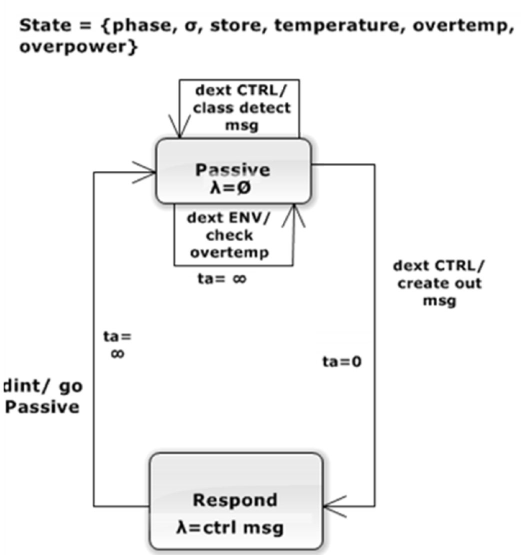

State = {phase, σ, store, temperature, overtemp, overpower}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 235. TPG Controller DEVS phase transition diagram

Z.6 TPG Controller Event-Trace Diagram

This section shows various examples of messages entering the controller. The tables list the states the component proceeds through as the events are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the component, the over temperature flag variable and the over power flag variable. The queue column shows the contents of the queue at that state, the contents of the store state variable and any notes. Note in contrast to most other components, the controller is very simple and only responds to incoming messages; it does not generate any messages on its own. There are two types of inputs: control messages and environmental messages.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Notes: any notes for that state

Z.6.1 CASE I: Initial Passive with Single Control Packet Arriving at Time 0

Table 99. *Case I state list*.

time	**state**	**entry/ exit**	**phase**	**sigma**	**store (*xi*)**	**temp**	**over temp**	**over power**	**Notes: assume tp=0**
	1-packet	no env	no ext	1 ctrl					
0	s0	entry	passive	inf	null	c	n	n	
0	s0	exit	passive	0	null	c	n	n	
0	s1	entry	respond	0	null	c	n	n	
0	s1	exit	respond	inf	null	c	n	n	
0	s2	entry	passive	inf	null	c	n	n	

Z.6.2 CASE II: Initial Passive with Single Environmental Packet Arriving at Time 0

Table 100. *Case II state list*.

		entry/			store		over	over	Notes:
time	state	exit	phase	sigma	(*xi*)	temp	temp	power	assume tp=0
	1-packet	1 env	no ext	0 ctrl					
0	s0	entry	passive	inf	null	c	n	n	
0	s0	exit	passive	0	null	c	n	n	
0	S1	entry	passive	inf	null	c	n	n	

Z.7 TPG Controller Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event

For the controller we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \text{ x } X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \text{ x } X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

$DEVS_{TPGcontroller} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

where

t_p = transmission time inside the component
temperature = current temperature of the component
phase = control state that keeps track of the internal phase of the component
phase = {“passive”, “respond”}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
interruptRespond = flag variable set when Respond phase is interrupted by an external event
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
ctrlOutput = variable that stores the output control message response
output.port = variable that holds the output optical packet port
store = variable that holds values of the current input values
timeLeftRespond = time left in Respond phase for the current event
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
ctrlMsg() = method that generates a response message to received control messages
messagebag_first() = method that returns the first element of the message bag
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {“$CtrlIn_1$”, “$CtrlIn_2$” “EnvIn”} with
X_M = {(“$CtrlIn_1$”, V_{ctrl}), (“$CtrlIn_2$”, V_{ctrl}), (“EnvIn”, V_{env})} is the set of input ports and values.

OutPorts = {“$CtrlOut_1$”, “$CtrlOut_2$”} with
Y_M = {(“$CtrlOut_1$”, $Y_{CtrlOut1}$), (“$CtrlOut_2$”, $Y_{CtrlOut2}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower*} = {{“passive”, “respond”} x R_0^+ x *V* x *R* x {“Y”, “N”} x {“Y”,”N”}}

External Transition Function:
δ_{ext}(*phase*, *σ*, *store, temperature*, *overtemp*, *overpower,e*, ((p_i,v_i),…. (p_n,v_n))) =
(“respond”, 0, *store, temperature*, *overtemp*, *overpower*)
  if *phase* = “passive” and *p* = “$CtrlIn_1$”

```
    ctrlOutput = ctrlMsg(store)
    if ctrlMsg.status = “init” or “get status”
      outputPort = “$\text{CtrlOut}_1$”
    if ctrlMsg.status = “fire laser”
      outputPort = “$\text{CtrlOut}_2$”

(“passive”, $\infty$, store, temperature, overtemp, overpower)
   if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

(phase, $\sigma - e$, store, temperature, overtemp, overpower)
        otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower) =
  (“passive”, $\infty$, store, temperature, overtemp, overpower)
   if phase = “respond”
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower) =
  (outputPort, ctrlOutput)
     if phase = “respond”

        $\emptyset$ (null output)
    otherwise;
```

Time advance Function:

ta(*phase*, σ, *store, temperature*, *overtemp*, *overpower*) = σ;

Z.8 TPG Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.
- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

For the TPG compound module we define:

Parallel-DEVS *compound* $N= (X, Y, D, \{M_d \mid \mathrm{d} \in D\}, EIC, EOC, IC)$

Where:

$X = \{(p,v) \mid \mathrm{p} \in IPorts,\ v \in X_p\}$ is the set of input ports and values;
$Y = \{(p,v) \mid \mathrm{p} \in OPorts,\ v \in Y_p\}$ is the set of output ports and values;
D = set of component names;
$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ is a DEVS atomic model;
$X_d = \{(p,v) \mid \mathrm{p} \in IPorts,\ v \in X_p\}$;
$Y_d = \{(p,v) \mid \mathrm{p} \in OPorts,\ v \in Y_p\}$;
$EIC \subseteq \{((N, ip_N),(d,ip_d)) \mid ip_N \in IPorts,\ d \in D,\ ip_d \in Iports_d\}$;
$EOC \subseteq \{((d,op_d),(N,op_N)) \mid op_N \in OPorts,\ d \in D,\ op_d \in Oports_d\}$;
$IC \subseteq \{((a,op_a),(b,ip_b)) \mid a,b \in D,\ op_a \in Oports_a,\ ip_b \in Iports_b\}$;
$((d,op_d),(e,ip_d)) \in IC$ implies $d \neq e$ (no feedback loops);
M = an atomic instance of P-DEVS.
N = a compound instance of P-DEVS.

***DEVS$_{TPG}$* = (*X, Y, D, {M$_d$* | d *∈D*}, *EIC*, *EOC*, *IC*)**

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$”, “OptIn$_1$”, “OptIn$_2$”, “OptIn$_3$”, “EnvIn”}
X = {(“CtrlIn$_1$”, *v*), (“CtrlIn$_2$”, *v*), (“OptIn$_1$”, *v*), (“OptIn$_2$”, *v*), (“OptIn$_3$”, *v*), (“EnvIn”, *v*) |$v \in V$}

OutPorts = {“CtrlOut$_1$”, “OptOut$_1$”, “OptOut$_2$”, “OptOut$_3$”}
Y = {(“CtrlOut$_1$”, *v*), (“OptOut$_1$”, *v*), (“OptOut$_2$”, *v*), (“OptOut$_3$”, *v*)|$v \in V$}

D = {controller, wdm, laser, polarizer, attenuator, SMfiber$_1$, SMfiber$_2$, SMfiber$_3$, SMfiber$_4$, SMfiber$_5$}
$M_d = M_{controller}, M_{wdm}, M_{laser}, M_{polarizer}, M_{attenuator}, Ms_{Mfiber1}, M_{SMfiber2}, Ms_{Mfiber3}, Ms_{Mfiber4}, Ms_{Mfiber5}$

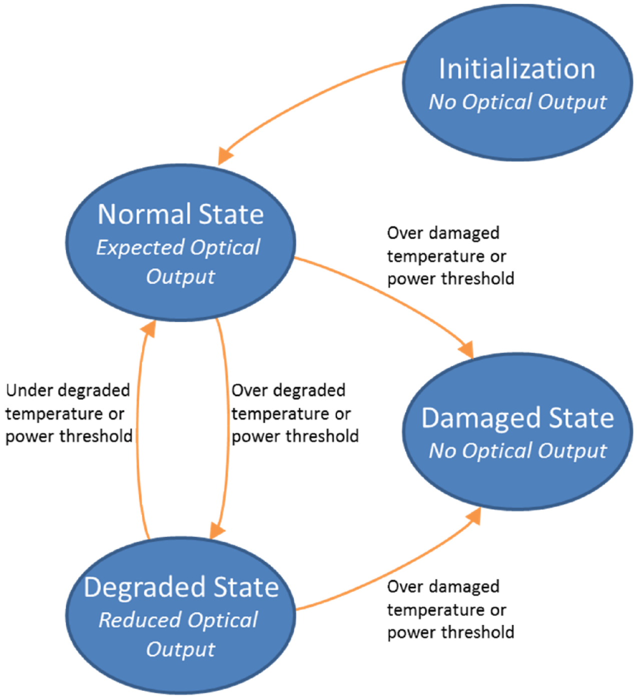
EIC = {((*N*, “CtrlIn$_1$”),(controller, “CtrlIn$_1$”)), ((*N*, “EnvIn”),(controller, “EnvIn”)), ((*N*, “EnvIn”),(wdm, “EnvIn”)), ((*N*, “EnvIn”),(laser, “EnvIn”)), ((*N*, “EnvIn”),(polarizer, “EnvIn”)), ((*N*, “EnvIn”),(attenuator, “EnvIn”)), ((*N*, “EnvIn”),(SMfiber$_1$, “EnvIn”)), ((*N*, “EnvIn”),(SMfiber$_2$, “EnvIn”)), ((*N*, “EnvIn”),(SMfiber$_3$, “EnvIn”)),(SMfiber$_4$, “EnvIn”)), (SMfiber$_5$, “EnvIn”)), ((*N*, “OptIn$_1$”),(SMfiber$_1$, “OptIn$_1$”)), ((*N*, “OptIn$_2$”),(SMfiber$_2$, “OptIn$_2$”))}

EOC = {((SMfiber$_1$, “OptOut$_1$”),(*N*, “OptOut$_1$”)), ((controller, “CtrlOut$_1$”),(*N*, “CtrlOut$_1$”)), ((SMfiber$_2$, “OptOut$_2$”),(*N*, “OptOut$_2$”))}

IC = {((laser, “CtrlOut$_1$”),(controller, “CtrlIn$_2$”)),
((SMfiber$_1$, “OptOut$_2$”),(wdm, “OptIn$_1$”)), ((wdm, “OptOut$_1$”),(SMfiber$_1$, “OptIn$_2$”)),
((wdm, “OptOut$_3$”),(SMfiber$_2$, “OptIn$_1$”)), ((SMfiber$_2$, “OptOut$_1$”),(wdm, “OptIn$_3$”)),
((laser, “OptOut$_1$”),(SMfiber$_5$, “OptIn$_1$”)), ((SMfiber$_5$, “OptOut$_1$”),(laser, “OptIn$_1$”)),
((SMfiber$_5$, “OptOut$_2$”),(polarizer, “OptIn$_1$”)), ((polarizer, “OptOut$_1$”),(SMfiber$_5$, “OptIn$_2$”)),

((polarizer, “OptOut_2”),(SMfiber_4, “OptIn_1”)), ((SMfiber_4, “OptOut_1”),(polarizer, “OptIn_2”)), ((SMfiber_4, “OptOut_2”),(attenuator, “OptIn_1”)), ((attenuator, “OptOut_1”),(SMfiber_4, “OptIn_2”)), ((attenuator, “OptOut_2”),(SMfiber_3, “OptIn_1”)), ((SMfiber_3, “OptOut_1”),(attenuator, “OptIn_2”)), ((SMfiber_3, “OptOut_2”),(wdm, “OptIn_2”)), ((wdm, “OptOut_2”),(SMfiber_3, “OptIn_2”))}

Z.9 TPG Controller Use Cases


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 236. Component states.

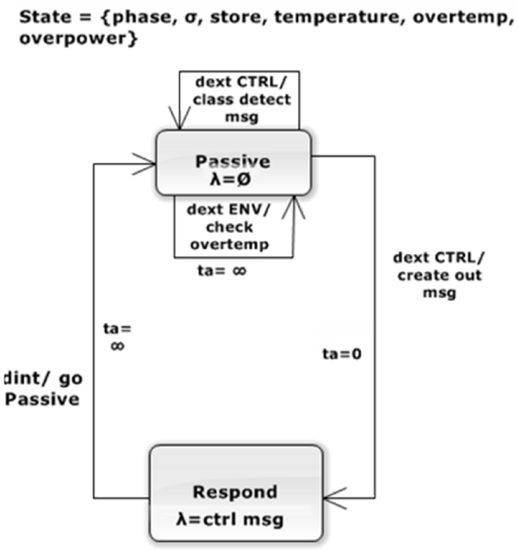

State = {phase, σ, store, temperature, overtemp, overpower}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 237. Controller phase transition diagram

Z.9.1 Respond to a Reset Message

Incoming reset message arrives at the module from the quantum controller. Pass the message to the module controller. Controller clears any stored variable values and prepares an acknowledgement message. Response message is sent out the appropriate port.

- Identified Alternative Uses Cases
  - React to an environmental message
  - React to a status request message
  - React to a fire laser message
  - React to a classical detector pulse detection message

- Assumptions
  - Incoming electrical signals are not affected by component state

Z.9.2 Respond to Reset Message End Goals

- Message properly received
- Controller enters Respond phase and sets storage values to zero.
- Controller forwards Reset Message to proper component(s) as necessary
- Acknowledgement message created and sent out the appropriate port
- Controller ends in Passive phase

Z.9.3 Respond to an Environmental Packet

Environmental packet arrives at the controller. Check to see if environmental packet temperature sets the controller to degraded or damaged state. Check to see if temperature level returns controller from degraded state to normal state. Records change in condition, if applicable. Change controller function if in degraded or damaged state, if necessary.

- Assumptions
    - None

Z.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

Z.9.5 Respond to a Status Request Message

Status Request message arrives at the module from the quantum controller. Module controller prepares response message. Response message is sent out the appropriate port.

- Assumptions
    - Controller has completed initialization sequence at least once

Z.9.6 Respond to Status Request End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

Z.9.7 Respond to a Fire Laser Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to laser component.

- Assumptions
    - Controller has completed initialization sequence at least once

Z.9.8 Respond to Fire Laser Message End Goals

- Fire laser message received properly

- Fire message recognized and passed to laser

Z.10 TPG Module Use Cases

Z.10.1 Respond to an Optical Packet

Optical packet arrives at the module. Pass the optical packet to the proper internal component.

- Assumptions
    - Reflections are not affected by module or component state

Z.10.2 Respond to Optical Packet End Goals

- Optical packet sent to proper internal component

Z.10.3 Respond to an Environmental Message

Environmental packet arrives at the module. Environmental message is passed to the module controller and each component in the module.

- Assumptions
    - Incoming electrical signals are not affected by component state

Z.10.4 Respond to Environmental Message End Goals

- Environmental packet received properly and forwarded to each component

Z.10.5 Respond to a Control Message

Control message arrives at the module. Control message is passed to the module controller.

- Assumptions
    - Incoming electrical signals are not affected by component state

Z.10.6 Respond to Environmental Message End Goals

- Control message received properly and forwarded to the module controller

Z.11 TPG Test Cases

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The type of control message passed to each coupled submodule depended on the internal components.

- TPG submodule – control message fires timing laser

These test cases led to iterations of testing and correction. Optical messages were tracked through the internal components and out the submodule output. Environmental messages were checked to ensure they replicated to each internal component. All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

	total tests	optical ports	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

Z.12 References

OZOptics. (2013). Wave division multiplexers. Retrieved, 2013, Retrieved from http://www.ozoptics.com/ALLNEW_PDF/DTS0089.pdf

Saleh, B. E. A., & Teich, M. C. (1991a). Guided waves. *Fundamentals of photonics* (2nd ed., pp. 340-342). New York: John Wiley & Sons, Inc.

Saleh, B. E. A., & Teich, M. C. (1991b). Laser diodes. *Fundamentals of photonics* (2nd ed., pp. 716-717). New York: John Wiley & Sons, Inc.

ThorLabs. (2013a). Coherent sources. Retrieved, 2013, Retrieved from http://www.thorlabs.com/navigation.cfm?guide_id=31

ThorLabs. (2013b). Fixed fiber optical attenuators. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1385

ThorLabs. (2013c). In-line fiber-optic polarizers. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=5922

ThorLabs. (2013d). Single-mode fiber. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=949

Appendix AA - Output Power Monitor (OPM)

AA.1 Device Description:

Alice needs a way to verify her output into the quantum channel. Components change as they age and some protocols may call for pulses of differing optical power. The output power monitor (OPM) allows Alice to sample the outgoing optical packets by using a photon detector capable of counting photons. By sampling the photon numbers, Alice can adjust the EVOAs to output the proper optical power. The switch alters the optical path by diverting optical packets to either the OPM or out of Alice into the quantum channel. .The Output Power Monitor subsystem contains the components shown in Fig. 1.

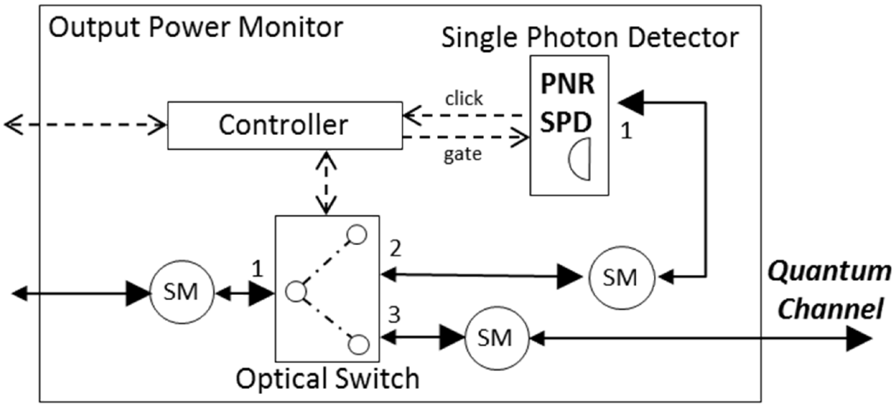


Figure 238. Output Power Monitor (OPM) in the QKD system architecture.

The OPM subsystem contains a controller, a photon number resolving single photon detector (PNR-SPD), an optical switch, electrical interfaces, and interconnecting SM optical fiber. We briefly discuss the behavior of each of the components contained within the OPM.

AA.1.1 OPM Controller

The controller is an electrical device containing digital and analog circuits responsible for controlling the single photon detector (SPD). It has a bidirectional electrical interface to the quantum module controller and an electrical output to the SPD. It receives commands from the quantum model controller, receives information from the SPD, and monitors the health of the SPD.

AA.1.2 Photon Number Resolving Single Photon Detector (PNR-SPD)

The PNR-SPD is an opto-electrical device containing detection equipment and support electronics capable of counting individual photons. The PNR-SPD has a single bidirectional optical port and a bidirectional electrical port connected to the OPM controller

AA.1.3 Optical Switch

The optical switch is used to route light between one input port and two or more input/output ports. The optical switch is a bidirectional optical component with three optical ports. Optical signals arriving at one of the ports is directed to one of the two input/output ports and propagated to the other port after a defined propagation delay. Typically, optical switches have control interfaces that allow them to be mounted on circuit boards or have some other type of electrical control port (DiConFiberOptics, 2013; ThorLabs, 2013a). The switch output couples using SM fiber to either the PNR-SPD or the quantum channel.

AA.1.4 Single-Mode Optical Fiber

SM fiber is an optical component used to interconnect optical devices. It has two bidirectional optical ports. Optical signals arriving at one port propagate to the other port after a defined propagation delay with its attenuation a function of the type and the length of the fiber. It is a cylindrical optical waveguide made from a low-loss material, such as silica glass. It has a core which guides the light and an outer cladding that reflects the internal light back into the core, bouncing the light down the fiber. This cladding helps to reflect outside light to keep in from entering the core. This structure allows for low loss over long distances. The single-mode of the fiber comes from using a small core diameter (~10µm @ 1550nm) and small numerical aperture with the fundamental mode having a bell-shaped spatial distribution similar (Saleh & Teich, 1991; ThorLabs, 2013b). SM fiber couples devices within the module.

AA.2 OPM and Controller Behavior

The controller and individual components are sensitive to the temperature in the environment in which they operate. If the temperature exceeds defined thresholds, the components may become temporarily degraded or permanently damaged which changes their characteristics. If temporarily degraded, the devices may recover to normal operating behavior after the temperature returns to a "normal" operating temperature.

The first step involved with modeling the controller and module is to collect and understand the physical, behavioral, and performance characteristics of the atomic components. In this case, the individual components were constructed earlier and the controller was built as a message handler. The logic for the controller was based on the types of messages necessary for control of components inside the module.

Once completed, the DEVS model is passed to the Software Development team that created a behaviorally equivalent C++ model in the OMNeT++ simulation environment during construction of the demonstration simulation. Comparing the demonstration simulation and timing and behavior outputs of the MS4ME models is the final step in validation testing the DEVS model.

AA.3 OPM Compound Conceptual Model

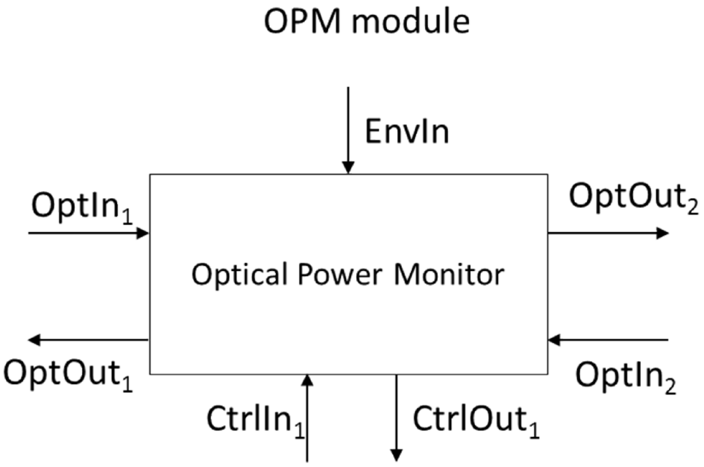


Figure 239. OPM compound module conceptual model.

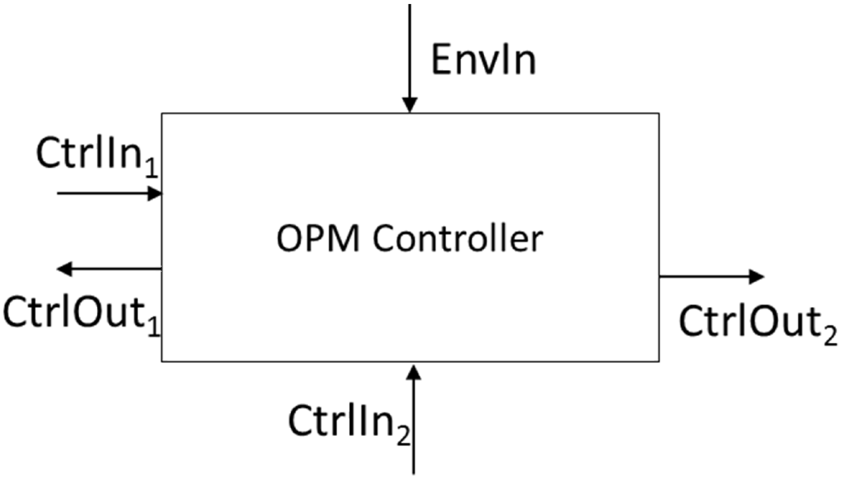


Figure 240. OPM controller conceptual model

Table 101. *List of OPM Controller messages.*

Input Messages	**From**	**Response**
OPM_ENV	Quantum controller	Set the internal controller temperature
OPM_RESET	Quantum controller	Resets the controller and clears the state variables
OPM_STATUS_REQUEST	Quantum controller	Sends the controller status
OPM_SET_SWITCH_PORT_2	Quantum controller	Set the switch to port 2
OPM_SET_SWITCH_PORT_3	Quantum controller	Set the switch to port 3

Output Messages	**To**	**Content**
OPM_ACK	Quantum Controller	Response to a Reset message
OPM_STATUS	Quantum Controller	Response to a Status Request message

The conceptual model for the OPM consists of two optical input ports {OptIn$_1$, OptIn$_2$}, two optical output ports {OptOut$_1$, OptOut$_2$}, one environmental input port {EvnIn}, one control input port {CtrlIn$_1$} and one control output port {CtrlOut$_1$}. The environmental port allows external sources to communicate changes in the operational environment to the module. The electrical controller ports allow for control inputs to the controller and responses from the module to the higher system functions.

In comparison to the module layout used in the QKD simulation architecture shown in Fig. 1, a single bidirectional optical connection is decomposed into an optical input and an optical output in the conceptual model. This is necessary to properly represent the behavior of the device using the DEVS formalism. The electrical control port is also decomposed in the model into an input port and an output port.

When an optical signal is sent to the input of the module, a small portion of the signal will be instantaneously reflected back to the signal source. Since the conceptual model decomposes each bidirectional connection to a discrete unidirectional output input and a discrete unidirectional optical output, this means that an optical signal arriving at OptIn$_1$ in Fig. 2 will instantaneously generate a reflected emitting out of OptOut$_1$.

The module components must calculate the power of each incoming optical signal in order to determine if the device will become damaged due to excessive power levels. This calculation is made when the packet first enters each of the components the module. In the case of optical overpowering, once overpowered a component will permanently change attenuation.

External environmental messages sent to the module are directed to individual components to convey the temperature of the operational environmental so the module can determine if it is degraded (a temporary condition) or damaged (a permanent condition). Changes to components based on the temperature determine the behavior of the module.

When multiple optical signals arrive at a port at the same time, they will be processed each as independent signals. This is a consequence of the high level simulation strategy to only model interference at the Single Photon Detector (SPD) devices in the QKD system simulation. This greatly simplifies the modeling of all of the other optical components which can treat multiple optical signals as independent entities.

AA.4 English-Language Rules for the Controller

In this section, English language rules are developed to express the desired behavior of the controller.

- CurrentTemp stores the current temperature. Initially, this is set to 25 degrees Centigrade.

- OverTemp is a flag which indicates if the device is permanently damaged due to being exposed to temperatures which exceed a defined temperature threshold. Initially, this flag is cleared.

When a control signal arrives:

- Determine the arrival port of the signal.
- Evaluate the content of the message
- Generate a response message to the incoming signal (if necessary).
- Generate a forwarded message to the appropriate device (if necessary).
- Output the response or forwarded message out the appropriate port.

When an environmental message arrives:

- Update the CurrentTemp with the current temperature contained in the environmental message.
- If the current temperature exceeds the damage temperature threshold, set the OverTemp flag.

AA.5 DEVS Phase Transition Diagram

The phase transition diagram in Fig. 4 shows the phases of the module controller in the boxes and the transitions represented by arrows between the phases. Each transition is labeled with the type of transition (d_{ext} – external or d_{int} – internal) and the significant actions that take place during the transition. Each arc has an entry either beneath or beside the arc indicating the value of the *time advance* function for the next phase. Each box is labeled with the name of the phase and an entry showing either no lambda output function for that phase or what the phase outputs.

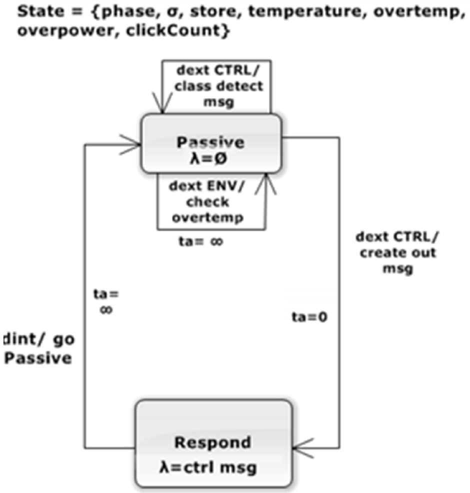

State = {phase, σ, store, temperature, overtemp, overpower, clickCount}
dext CTRL/ class detect msg
Passive
λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
ta=0
dint/ go Passive
Respond
λ=ctrl msg

Figure 241. OPM Controller DEVS phase transition diagram

AA.6 OPM Controller Event-Trace Diagram

This section shows various examples of messages entering the controller. The tables list the states the component proceeds through as the events are processed. Each table has the state number, with each state consisting of: phase, time until next transition (sigma), store state variable, current temperature of the component, the over temperature flag variable and the over power flag variable. The queue column shows the contents of the queue at that state, the contents of the store state variable and any notes. Note in contrast to most other components, the controller is very simple and only responds to incoming messages; it does not generate any messages on its own. There are two types of inputs: control messages and environmental messages.

Explanations for each column:

- Time: elapsed time since beginning of the case
- State: shows the state number starting with s0, the start state
- Phase: shows the phase for that state
- Sigma: the time until next internal transition. A 0 sigma indicates a transitory state
- Store: contents of the store variable for that state
- Temp: value of the current internal temperature. In this case, always some degree C value
- Over Temp: shows the value of the over temperature flag variable
- Over Power: shows the value of the over power flag variable
- Click Count: the number of clicks (photons) counted by the PNR SPD
- Notes: any notes for that state

AA.6.1 CASE I: Initial Passive with Single Control Packet Arriving at Time 0

Table 102. *Case I state list.*

		entry/			**store**		**over**	**over**		**Notes:**
time	**state**	**exit**	**phase**	**sigma**	**(*xi*)**	**temp**	**temp**	**power**	**click count**	**assume tp=0**
	1-packet	no env	no ext	1 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	
0	s1	entry	respond	0	null	c	n	n	null	
0	s1	exit	respond	inf	null	c	n	n	null	
0	s2	entry	passive	inf	null	c	n	n	null	

AA.6.2 CASE II: Initial Passive with Single Environmental Packet Arriving at Time 0

Table 103. *Case II state list.*

		entry/			**store**		**over**	**over**		**Notes:**
time	**state**	**exit**	**phase**	**sigma**	**(*xi*)**	**temp**	**temp**	**power**	**click count**	**assume tp=0**
	1-packet	1 env	no ext	0 ctrl						
0	s0	entry	passive	inf	null	c	n	n	null	
0	s0	exit	passive	0	null	c	n	n	null	

0	S1	entry	passive	inf	null	c	n	n	null	

AA.7 OPM Controller Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

Definitions:

State = {phase, time advance, “store”, temperature, “overtemp”, “overpower”, “currentAttenuation”}
Time advance(state) = time advance of the current state
Time delay = time advance stored in queue for event *i*
e = elapsed time since last transition occurred
“store” = state variable that stores the current input values
“overtemp” = flag variable set when device meets or exceeds damaged temperature level
“overpower” = flag variable set when device meets or exceeds damaged optical power level
“interruptRespond” = flag variable set when device is interrupted by an external event
“clickCount” = variable to store the current number of photons counted by the PNR SPD

For the controller we define:

Parallel-DEVS *atomic* $M = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$

Where:

$X_M = \{(p,v) \mid \mathrm{p} \in InPorts, v \in X_p\}$ is the set of input ports and values;
$Y_M = \{(p,v) \mid \mathrm{p} \in OutPorts, v \in Y_p\}$ is the set of output ports and values;
S = set of sequential states;
$\delta_{ext} = Q \times X_M^b \rightarrow S$ is the external state transition function;
$\delta_{int} = S \rightarrow S$ is the internal state transition function;
$\delta_{con} = Q \times X_M^b \rightarrow S$ is the confluent transition function;
$\lambda = S \rightarrow Y^b$ is the output function;
$ta = S \rightarrow R_0^+ \cup \infty$ or $S \rightarrow R_{0^+ \rightarrow \infty}$ is the time advance function;
$Q := \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total set of states;

X_b = a set of bags over elements of X;
M = an atomic instance of P-DEVS.

$\boldsymbol{DEVS_{OPMcontroller} = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)}$

where

t_p = transmission time inside the component
temperature = current temperature of the component
phase = control state that keeps track of the internal phase of the component
phase = {"passive", "respond"}
overtemp = flag variable set when device meets or exceeds damaged temperature level
overpower = flag variable set when device meets or exceeds damaged optical power level
clickCount = variable that holds the number of photons
interruptRespond = flag variable set when Respond phase is interrupted by an external event
messagebag= variable that stores the current *x* input value(s) (*p*,*v)*
damage.temp = variable that holds the component damaged temperature level parameter
current = variable that stores the queue event being manipulated
ctrlOutput = variable that stores the output control message response
output.port = variable that holds the output optical packet port
store = variable that holds values of the current input values
timeLeftRespond = time left in Respond phase for the current event
e = elapsed time since last transition occurred
σ = state variable that holds the time to next transition
ctrlMsg() = method that generates a response message to received control messages
messagebag_first() = method that returns the first element of the message bag
remove_event_m() = method that remove the current (x_i, time $delay_i$) from *messagebag*

Every δ_{ext} puts all of its *x* (p,v) values into the variable *store*

InPorts = {"$CtrlIn_1$", "$CtrlIn_2$", "$CtrlIn_3$", "EnvIn"} with
X_M = {("$CtrlIn_1$", V_{ctrl}), ("$CtrlIn_2$", V_{ctrl}), ("$CtrlIn_3$", V_{ctrl}), ("EnvIn", V_{env})} is the set of input ports and values.

OutPorts = {"$CtrlOut_1$", "$CtrlOut_2$", "$CtrlOut_3$"} with
Y_M = {("$CtrlOut_1$", $Y_{CtrlOut1}$), ("$CtrlOut_2$", $Y_{CtrlOut2}$), ("$CtrlOut_3$", $Y_{CtrlOut3}$)} is the set of output ports and values.

phase is a control state used to keep track of where the full state is.

S = {*phase*, σ, *store*, *temperature*, *overtemp*, *overpower, clickCount* } = {{"passive", "respond"} x R_0^+ x *V* x *R* x {"Y", "N"} x {"Y","N"} x *V* }

External Transition Function:

```
$\delta_{ext}$(phase, σ, store, temperature, overtemp, overpower, clickCount ,e, (($p_i$,$v_i$),…. ($p_n$,$v_n$))) =
("respond", 0, store, temperature, overtemp, overpower, clickCount)
  if phase = "passive" and p = "$CtrlIn_1$"
    ctrlOutput = ctrlMsg(store)
    if ctrlMsg.status = "init" or "get status"
```

```
      outputPort = “$\text{CtrlOut}_1$”
    if ctrlMsg.status = “set gate”
      outputPort = “$\text{CtrlOut}_2$”
    if ctrlMsg.status = “set port”
      outputPort = “$\text{CtrlOut}_3$”

(“passive”, 0, store, temperature, overtemp, overpower, clickCount)
  if phase = “passive” and p = “$\text{CtrlIn}_3$”
    clickCount = messagebag.count

(“passive”, $\infty$, store, temperature, overtemp, overpower, clickCount)
   if phase = “passive” and p = “EnvIn”
    temperature = messagebag.temperature
    if temperature > damage.temp
      overtemp = “Y”

(phase, $\sigma - e$, store, temperature, overtemp, overpower, clickCount)
  otherwise;
```

Internal Transition Function:

```
$\delta_{int}$(phase, $\sigma$, store, temperature, overtemp, overpower, clickCount) =
  (“passive”, $\infty$, store, temperature, overtemp, overpower, clickCount)
    if phase = “respond”
```

Confluence Function:

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

Output Function:

```
$\lambda$(phase, $\sigma$, store, temperature, overtemp, overpower, clickCount) =
  (outputPort, ctrlOutput)
    if phase = “respond”

  $\emptyset$ (null output)
    otherwise;
```

Time advance Function:

ta(*phase*, σ, *store, temperature*, *overtemp*, *overpower*, *clickCount*) = σ;

AA.8 OPM Parallel DEVS Code

Notes:

- Assume that only one environmental packet will arrive at any given time, due to the small time scales involved and the length of time necessary for temperature fluctuations.

- The component will always reflect a portion of any incoming optical packet, regardless of the environmental state, discussions with the optical SMEs.
- If multiple optical packets arrive at the same time, they will be processed through the reflection state as a group, but then input into the queue as single entries with the same delay time.
- The reflection function always reflects the optical packet back out the port it arrived on.

For the OPM compound module we define:

Parallel-DEVS *compound* $N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$

Where:

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;
$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;
D = set of component names;
$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ is a DEVS atomic model;
$X_d = \{(p,v) \mid p \in IPorts, v \in X_p\}$;
$Y_d = \{(p,v) \mid p \in OPorts, v \in Y_p\}$;
$EIC \subseteq \{((N, ip_N),(d,ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in Iports_d\}$;
$EOC \subseteq \{((d,op_d),(N,op_N)) \mid op_N \in OPorts, d \in D, op_d \in Oports_d\}$;
$IC \subseteq \{((a,op_a),(b,ip_b)) \mid a,b \in D, op_a \in Oports_a, ip_b \in Iports_b\}$;
$((d,op_d),(e,ip_d)) \in IC$ implies $d \neq e$ (no feedback loops);
M = an atomic instance of P-DEVS.
N = a compound instance of P-DEVS.

$$\boldsymbol{DEVS_{OPM} = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)}$$

InPorts = {“CtrlIn$_1$”, “CtrlIn$_2$”, “OptIn$_1$”, “OptIn$_2$”, “EnvIn”}
X = {(“CtrlIn$_1$”, v), (“CtrlIn$_2$”, v), (“OptIn$_1$”, v), (“OptIn$_2$”, v), (“EnvIn”, v) |$v \in V$}

OutPorts = {“CtrlOut$_1$”, “CtrlOut$_2$”, “OptOut$_1$”, “OptOut$_2$”}
Y = {(“CtrlOut$_1$”, v), (“CtrlOut$_2$”, v), (“OptOut$_1$”, v), (“OptOut$_2$”, v) |$v \in V$}

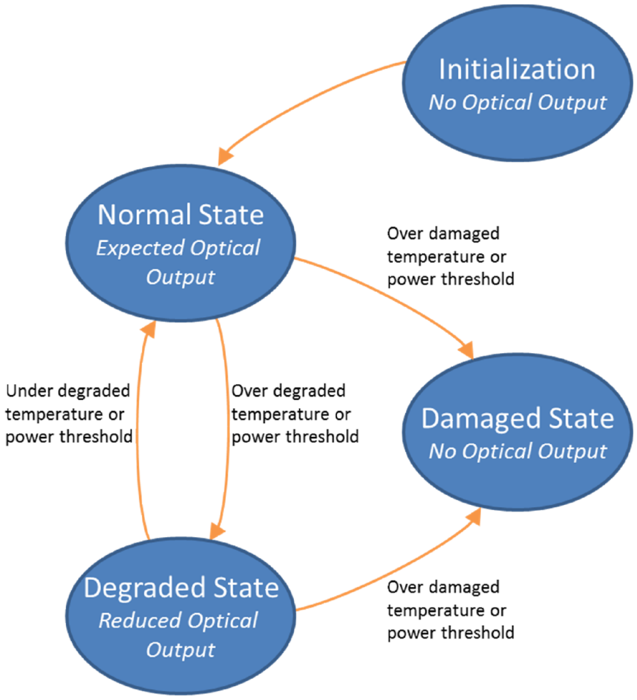
D = {controller, switch, pnrspd, SMfiber$_1$, SMfiber$_2$, SMfiber$_3$}
$M_d = M_{controller}, M_{switch}, M_{pnrspd}, M_{SMfiber1}, M_{SMfiber2}, M_{SMfiber3}$

EIC = {((N, “CtrlIn$_1$”),(controller, “CtrlIn$_1$”)), ((N, “EnvIn”),(controller, “EnvIn”)), ((N, “EnvIn”),(switch, “EnvIn”)), ((N, “EnvIn”),(pnrspd, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_1$, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_2$, “EnvIn”)), ((N, “EnvIn”),(SMfiber$_3$, “EnvIn”)), ((N, “OptIn$_1$”),(SMfiber$_1$, “OptIn$_1$”)), ((N, “OptIn$_2$”),(SMfiber$_2$, “OptIn$_2$”))}

EOC = {((SMfiber$_1$, "OptOut$_1$"),(*N*, "OptOut$_1$")), ((controller, "CtrlOut$_1$"),(*N*, "CtrlOut$_1$")), ((SMfiber$_2$, "OptOut$_2$"),(*N*, "OptOut$_2$"))}

IC = {((controller, "CtrlOut$_3$"),(switch, "CtrlIn$_1$")), ((switch, "CtrlOut$_1$"),(controller, "CtrlIn$_3$")), ((controller, "CtrlOut$_2$"),(pnrspd, "CtrlIn$_1$")), ((pnrspd, "CtrlOut$_1$"),(controller, "CtrlIn$_2$")), ((SMfiber$_1$, "OptOut$_2$"),(switch, "OptIn$_1$")), ((switch, "OptOut$_1$"),(SMfiber$_1$, "OptIn$_2$")), ((switch, "OptOut$_2$"),(SMfiber$_3$, "OptIn$_1$")), ((SMfiber$_3$, "OptOut$_1$"),(switch, "OptIn$_2$")), ((switch, "OptOut$_3$"),(SMfiber$_2$, "OptIn$_1$")), ((SMfiber$_2$, "OptOut$_1$"),(switch, "OptIn$_3$")), ((SMfiber$_3$, "OptOut$_2$"),(pnrspd, "OptIn$_1$")), ((pnrspd, "OptOut$_1$"),(SMfiber$_3$, "OptIn$_2$"))}

AA.9 OPM Controller Use Case


Initialization
No Optical Output
Normal State
Expected Optical Output
Over damaged temperature or power threshold
Under degraded temperature or power threshold
Over degraded temperature or power threshold
Damaged State
No Optical Output
Degraded State
Reduced Optical Output
Over damaged temperature or power threshold

*Reflections are not configured to be effected by state
*Electrical signals are not configured to be effected by state

Figure 242. Component states.

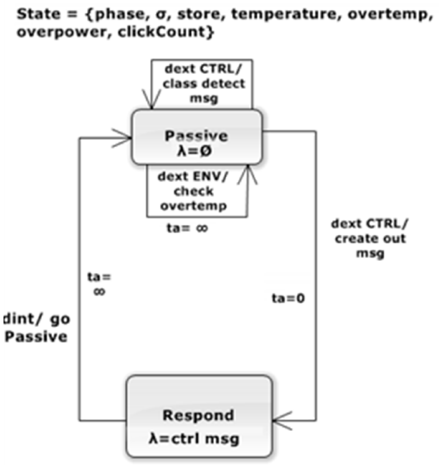

State = {phase, σ, store, temperature, overtemp, overpower, clickCount}
dext CTRL/ class detect msg
Passive λ=Ø
dext ENV/ check overtemp
ta= ∞
dext CTRL/ create out msg
ta= ∞
dint/ go Passive
ta=0
Respond λ=ctrl msg

Figure 243. Controller phase transition diagram

AA.9.1 Respond to a Reset Message

Incoming reset message arrives at the module from the quantum controller. Pass the message to the module controller. Controller clears any stored variable values and prepares an acknowledgement message. Response message is sent out the appropriate port.

- Identified Alternative Uses Cases
  - React to an environmental message
  - React to a status request message
  - React to a fire laser message
  - React to a classical detector pulse detection message

- Assumptions
  - Incoming electrical signals are not affected by component state

AA.9.2 Respond to Reset Message End Goals

- Message properly received
- Controller enters Respond phase and sets storage values to zero.
- Controller forwards Reset Message to proper component(s) as necessary
- Acknowledgement message created and sent out the appropriate port
- Controller ends in Passive phase

AA.9.3 Respond to an Environmental Packet

Environmental packet arrives at the controller. Check to see if environmental packet temperature sets the controller to degraded or damaged state. Check to see if temperature level returns controller from degraded state to normal state. Records change in condition, if applicable. Change controller function if in degraded or damaged state, if necessary.

- Assumptions
    - None

AA.9.4 Respond to Environmental Packet End Goals

- Environmental packet received properly
- Overtemperature condition properly recognized and recorded
- Change of state completed and recorded properly, if necessary
- Change component function properly, if necessary

AA.9.5 Respond to a Status Request Message

Status Request message arrives at the module from the quantum controller. Module controller prepares response message. Response message is sent out the appropriate port.

- Assumptions
    - Controller has completed initialization sequence at least once

AA.9.6 Respond to Status Request End Goals

- Control message received properly
- Change of condition or state completed and recorded properly, if necessary
- Change component function properly, if necessary

AA.9.7 Respond to a Set Switch Port 2 Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
    - Controller has completed initialization sequence at least once

AA.9.8 Respond to Set Switch Port 2 Message End Goals

- Set Switch Port 2 message received properly
- Message recognized and passed to the proper component

AA.9.9 Respond to a Set Switch Port 3 Message

Incoming control message arrives at the module from the quantum controller. Pass the message to the module controller. Module controller passes control message to the proper component.

- Assumptions
  - Controller has completed initialization sequence at least once

AA.9.10 Respond to Set Switch Port 3Message End Goals

- Set Switch Port 3 message received properly
- Message recognized and passed to the proper component

AA.10 OPM Module Use Cases

AA.10.1 Respond to an Optical Packet

Optical packet arrives at the module. Pass the optical packet to the proper internal component.

- Assumptions
  - Reflections are not affected by module or component state

AA.10.2 Respond to Optical Packet End Goals

- Optical packet sent to proper internal component

AA.10.3 Respond to an Environmental Message

Environmental packet arrives at the module. Environmental message is passed to the module controller and each component in the module.

- Assumptions
  - Incoming electrical signals are not affected by component state

AA.10.4 Respond to Environmental Message End Goals

- Environmental packet received properly and forwarded to each component

AA.10.5 Respond to a Control Message

Control message arrives at the module. Control message is passed to the module controller.

- Assumptions
  - Incoming electrical signals are not affected by component state

AA.10.6 Respond to Environmental Message End Goals

- Control message received properly and forwarded to the module controller

AA.11 OPM Test Cases

Each coupled submodule was tested by sending messages to the submodule and using the operational graphics of the MS4ME simulator to track the progress of the message through the submodule. The primary purpose of the test cases was testing the ability of the coupled submodule to receive messages, pass them internally to the submodule controller and pass internal output to external ports. The controller processed these input messages and passed an appropriate message to the controlled opto-electrical component. The type of control message passed to each coupled submodule depended on the internal components.

- OPM submodule – control message changes optical switch position

These test cases led to iterations of testing and correction. Optical messages were tracked through the internal components and out the submodule output. Environmental messages were checked to ensure they replicated to each internal component. All the errors identified in the coupled submodules were problems with coding the controllers, as the atomic components functioned properly during coupling.

Table 4. *Summary of Coupled Submodule Behavior Testing.*

	total tests	optical ports	ctrl port	env port
Classical Pulse Generator	4	0	3	1
Polarization Modulator	5	1	3	1
Decoy State Generator	5	1	3	1
Classical To Quantum	5	1	3	1
Optical Security Layer	4	1	2	1
Timing Pulse Generator	5	1	3	1
Optical Power Monitor	5	1	3	1

AA.12 References

DiConFiberOptics. (2013). MEMS 1x2 switch. Retrieved, 2013, Retrieved from http://www.diconfiberoptics.com/products/scd0044/0044h.pdf

Saleh, B. E. A., & Teich, M. C. (1991). Guided waves. *Fundamentals of photonics* (2nd ed., pp. 340-342). New York: John Wiley & Sons, Inc.

ThorLabs. (2013a). MEMS fiber-optic switches. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1553

ThorLabs. (2013b). Single-mode fiber. Retrieved, 2013, Retrieved from http://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=949

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
18-09-2014

2. REPORT TYPE
Dissertation

3. DATES COVERED *(From - To)*
Sept 2011 - Sept 2014

4. TITLE AND SUBTITLE
Conceptual Modeling of a Quantum Key Distribution Simulation Framework Using the Discrete Event System Specification

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Morris, Jeffrey D., MSG, USA

5d. PROJECT NUMBER
14V137

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT-ENV-DS-14-S-25

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Laboratory for Telecommunication Sciences
Dr. Gerry Baumgartner
8080 Greenmead Drive
College Park MD 20740
gbaumgartner@ltsnet.net

10. SPONSOR/MONITOR'S ACRONYM(S)
LTS

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Statement A. Approved for Public Release;Distribution Unlimited

13. SUPPLEMENTARY NOTES
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Quantum Key Distribution (QKD) is a revolutionary security technology that exploits the laws of quantum mechanics to achieve information-theoretical secure key exchange. QKD is suitable for use in applications that require high security such as those found in certain commercial, governmental, and military domains. As QKD is a new technology, there is a need to develop a robust quantum communication modeling and simulation framework to support the analysis of QKD systems. This dissertation presents conceptual modeling QKD system components using the Discrete Event System Specification (DEVS) formalism to assure the component models are provably composable and exhibit temporal behavior independent of the simulation environment. These attributes enable users to assemble and simulate any collection of compatible components to represent QKD system architectures. The developed models demonstrate closure under coupling and exhibit behavior suitable for the intended analytic purpose, thus improving the validity of the simulation. This research contributes to the validity of the QKD simulation, increasing developer and user confidence in the correctness of the models and providing a composable, canonical basis for performance analysis efforts. The

15. SUBJECT TERMS
Conceptual Modeling; Discrete Event Simulation; Discrete Event System Specification; Quantum Key Distribution; Modeling and Simulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Michael R. Grimaila, AFIT/ENV
U	U	U	UU	769	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636 x4800 michael.grimaila@afit.edu

Reset